Guardian Procedure Calls Reference Manual

Abstract

This manual describes the syntax for most Guardian procedure calls. This manual is for programmers who need to call Guardian procedures from their programs.

Product Version

N.A.

Supported Release Version Updates (RVUs)

This publication supports J06.03 and all subsequent J-series RVUs, H06.03 and all subsequent H-series RVUs, and G06.27 and all subsequent G-series RVUs, until otherwise indicated by its replacement publications. Additionally, all considerations for H-series throughout this manual will hold true for J-series also, unless mentioned otherwise.

Part Number	Published
522629-030	August 2010

Document History

Part Number	Product Version	Published
522629-025	N.A.	February 2009
522629-026	N.A.	May 2009
522629-027	N.A	August 2009
522629-028	N.A.	February 2010
522629-029	N.A	May 2010
522629-030	N.A	August 2010

Guardian Procedure Calls Reference Manual

3. Guardian Procedure Calls (C)

5. Guardian Procedure Calls (F)

7. Guardian Procedure Calls (H-K)

8. Guardian Procedure Calls (L)

9. Guardian Procedure Calls (M)

12. Guardian Procedure Calls (P)

14. Guardian Procedure Calls (S)

Tables

What's New in This Manual

Manual Information

Abstract

This manual describes the syntax for most Guardian procedure calls. This manual is for programmers who need to call Guardian procedures from their programs.

Product Version

N.A.

Supported Release Version Updates (RVUs)

This publication supports J06.03 and all subsequent J-series RVUs, H06.03 and all subsequent H-series RVUs, and G06.27 and all subsequent G-series RVUs, until otherwise indicated by its replacement publications. Additionally, all considerations for H-series throughout this manual will hold true for J-series also, unless mentioned otherwise.

Part Number	Published
522629-030	August 2010

Document History

Part Number	Product Version	Published
522629-025	N.A.	February 2009
522629-026	N.A.	May 2009
522629-027	N.A	August 2009
522629-028	N.A.	February 2010
522629-029	N.A	May 2010
522629-030	N.A	August 2010

New and Changed Information

Changes to the H06.21/J06.10 manual:

- Updated description of *keyspecifier* parameter of FILE_SETKEY_ procedure on page 5-138 .
- Added the following procedures:
  - STACK_ALLOCATE_ Procedure on page 14-150.
  - STACK_DEALLOCATE_ Procedure on page 14-154.

- Updated information about Running the DIVER Program on page I-1 and Example Using DIVER and DELAY on page I-3.

Changes to the 522629-029 manual:

- Added two warning notes about modifying buffers on page 2-48.
- Updated the *sync-or-receive-depth* parameter on page 5-114.
- Added a consideration to the FILE_OPEN procedure on page 5-124.
- Updated the *sync-or-receive-depth* parameter on page 11-15.
- Added a consideration to the FILE_OPEN procedure on page 11-24.
- Updated Attribute 82 for Table 12-2, PROCESS_GETINFOLIST_ Attribute Codes and Value Representations, on page 12-75.
- Added a warning note about on modifying buffers for the nowait file on page 13-5, 13-21, 13-27, 13-35, 16-8, 16-22, 16-27, and 16-34.
- Updated Table 14-4, SETMODE Functions with the SETMODE 266 function on page 14-99.
- Added a disk subtype on page A-2.
- Added a caution note about running the DIVER program after RELOAD on page I-1.
- Updated the DELAY timings in the example on page I-3.
- Updated Table J-3, Per-Processor Limits on page J-5.
- Updated Table J-5, Enscribe File System Limits on page J-7.

Changes to the H06.20/J06.09 Manual

- Added Attribute 79 for Processor_GETINFOLIST on 12-229 and its description on 12-235.
- Updated the syntax for `short *finished-length` in the following procedures:
  - MBCS_EXTERNAL_TO_TANDEM_ Procedure on page 9-14.
  - MBCS_TANDEM_TO_EXTERNAL_ Procedure on page 9-36.
- Updated information about *maximum-length* on page 9-15 and page 9-38.
- Updated the Considerations section for the following procedures:
  - MBCS_EXTERNAL_TO_TANDEM_ Procedure on page 9-18.
  - MBCS_TANDEM_TO_EXTERNAL_ Procedure on page 9-41.
  - PROCESS_STOP_ Procedure on page 12-190.

- Removed a note from ADDDSTTRANSITION Procedure (Superseded by DST_GETINFO_ Procedure) on page 2-10.

Changes to the H06.19/J06.08 Manual

- Updated the attribute of Code 60 of PROCESSOR_GETINFOLIST_ Procedure on page 12-228.
- Updated general considerations of USER_AUTHENTICATE_ Procedure on pages 15-38 and 15-39.

Changes to the H06.18/J06.07 Manual

- Updated AWAITIO[X] procedure to AWAITIO[X|XL] procedure on page 2-40.
- Updated CANCELREQ procedure to CANCELREQ[L] procedure on page 3-6.
- Updated FILE_COMPLETE_ procedure to FILE_COMPLETE[L]_ procedure on page 5-16.
- Updated FILE_GETRECEIVEINFO_ procedure to FILE_GETRECEIVEINFO[L]_ procedure on page 5-104.
- Updated general considerations of FILE_OPEN procedure on page 5-120.
- Updated general considerations of OPEN procedure on page 11-21.
- Updated READUPDATE[X] procedure to READUPDATE[X|XL] procedure on page 13-23.
- Updated REPLY[X] procedure to REPLY[X|XL] procedures on page 13-53.
- Updated description of SETMODE function 72 on page 14-80.
- Updated considerations for SETMODENOWAIT Procedure on page 14-104.
- Updated description of USERIOBUFFER_ALLOW_ procedure on page 15-54.
- Added a new entry, files per volume, in the DP2 Limits table on page J-9.
- Updated H-series version in the DP2 Limits table on page J-9.

Changes to the H06.17/J06.06 Manual

- Added error return code 4002 and its description for BINSEM_CREATE_Procedure on page 2-55.
- Updated the description of error return code 4024 for BINSEM_CREATE_Procedure on page 2-55.
- Added description of BINSEM_ISMINE_ under Binary Semaphore Operations on page 2-57.
- Updated the description of Binary semaphore resource requirements on page 2-57.

- Added information on BINSEM_ISMINE_Procedure on page 2-60.
- Added a note in the considerations section of the BINSEM_LOCK_Procedure on page 2-63.
- Updated the description of error return code 4002 for BINSEM_OPEN_Procedure on page 2-65.
- Updated the description of error return code 4024 for BINSEM_OPEN_Procedure on page 2-65.
- Added a new processor type and model under Table 12-6, Summary of Processor Types and Models, on page 12-240.
- Updated Table 12-6, Summary of Processor Types and Models, on pages 12-237 and 12-238.
- Added the maximum value of Open user semaphore under Per-Process Limits table on page J-3.
- Added the maximum value of Open user semaphore under Per-Processor Limits table on page J-4.

About This Manual

This reference manual describes the syntax of most of the Guardian procedure calls.

Readership of This Manual

This manual is for programmers who need to call Guardian procedures from their programs. Familiarity with TAL or some other programming language is recommended.

Organization of This Manual

Section	Description
Section 1, Introduction to Guardian Procedure Calls	Gives an overview of the procedure calls and describes the format of a procedure call description
Sections 2 through 16	Describes the Guardian procedure calls in alphabetic order
Appendix A, Device Types and Subtypes	Lists the device types and subtypes (such as disks, printers, terminals, and so on) that are referred to by Guardian procedure calls
Section B, Reserved Process Names	Lists the process names reserved for use by HP
Appendix C, Completion Codes	Lists the completion codes returned after execution of a process that indicate, in a standard manner, its degree of success
Appendix D, File Names and Process Identifiers	Describes reserved file names, C-series and D-series syntax for file names and process identifiers, and the syntax for OSS pathnames
Appendix E, DEFINEs	Describes DEFINEs and lists the attributes of all classes of DEFINEs
Appendix F, Formatter Edit Descriptors	Describes the edit descriptors that are used by the formatter
Appendix G, Superseded Guardian Procedure Calls and Their Replacements	Lists the superseded Guardian procedure calls
Appendix H, Documented Guardian Procedures	Lists all documented Guardian procedures and the manuals in which they are documented
Appendix I, Using the DIVER and DELAY Programs	Describes the DIVER and DELAY programs
Appendix J, System Limits	Summarizes the architectural and programmatic limits that apply on HP NonStop™ servers
Appendix K, Character Set Translation	Lists an ASCII-EBCDIC translation

Related Manuals

While using this manual, you will find these manuals helpful:

- *Guardian Programmer's Guide*
- *Open System Services Programmer's Guide*
- *pTAL Reference Manual*
- *pTAL Conversion Guide*
- *TAL Reference Manual*
- *C/C++ Programmer's Guide*
- *Common Run-Time Environment (CRE) Programmer's Guide*

Before converting your applications to TNS/R native applications, you should read:

- *TNS/R Native Application Migration Guide*

Before converting your applications to TNS/E native applications, you should read:

- *H-Series Application Migration Guide*

For a summary of procedure calls, interprocess messages, error codes, and other material presented in a quick reference format, obtain a copy of:

- *Guardian Programming Reference Summary*

"TNS" refers to the series of computers based on complex instruction-set computing (CISC) technology.

"TNS/R" refers to the series of computers based on reduced instruction-set computing (RISC) technology.

"TNS/E" refers to the series of computers based on the Itanium Processor (IPF) computing technology.

"Word" is used throughout this manual to refer to a 2-byte unit of memory.

Notation Conventions

Hypertext Links

Blue underline is used to indicate a hypertext link within text. By clicking a passage of text with a blue underline, you are taken to the location described. For example:

> This requirement is described under Backup DAM Volumes and Physical Disk Drives on page 3-2.

General Syntax Notation

This list summarizes the notation conventions for syntax presentation in this manual.

UPPERCASE LETTERS. Uppercase letters indicate keywords and reserved words. Type these items exactly as shown. Items not enclosed in brackets are required. For example:

```
MAXATTACH
```

lowercase italic letters. Lowercase italic letters indicate variable items that you supply. Items not enclosed in brackets are required. For example:

```
file-name
```

computer type. `Computer type` letters within text indicate C and Open System Services (OSS) keywords and reserved words. Type these items exactly as shown. Items not enclosed in brackets are required. For example:

```
myfile.c
```

italic computer type. *`Italic computer type`* letters within text indicate C and Open System Services (OSS) variable items that you supply. Items not enclosed in brackets are required. For example:

```
pathname
```

[ ] Brackets. Brackets enclose optional syntax items. For example:

```
TERM [\system-name.]$terminal-name

INT[ERRUPTS]
```

A group of items enclosed in brackets is a list from which you can choose one item or none. The items in the list can be arranged either vertically, with aligned brackets on each side of the list, or horizontally, enclosed in a pair of brackets and separated by vertical lines. For example:

```
FC [ num  ]
   [ -num ]
   [ text ]

K [ X | D ] address
```

{ } Braces. A group of items enclosed in braces is a list from which you are required to choose one item. The items in the list can be arranged either vertically, with aligned braces on each side of the list, or horizontally, enclosed in a pair of braces and separated by vertical lines. For example:

```
LISTOPENS PROCESS { $appl-mgr-name }
                  { $process-name  }

ALLOWSU { ON | OFF }
```

| Vertical Line. A vertical line separates alternatives in a horizontal list that is enclosed in brackets or braces. For example:

```
INSPECT { OFF | ON | SAVEABEND }
```

... Ellipsis. An ellipsis immediately following a pair of brackets or braces indicates that you can repeat the enclosed sequence of syntax items any number of times. For example:

```
M address [ , new-value ]...

[ - ] {0|1|2|3|4|5|6|7|8|9}...
```

An ellipsis immediately following a single syntax item indicates that you can repeat that syntax item any number of times. For example:

```
"s-char..."
```

Punctuation. Parentheses, commas, semicolons, and other symbols not previously described must be typed as shown. For example:

```
error := NEXTFILENAME ( file-name ) ;

LISTOPENS SU $process-name.#su-name
```

Quotation marks around a symbol such as a bracket or brace indicate the symbol is a required character that you must type as shown. For example:

```
"[" repetition-constant-list "]"
```

Item Spacing. Spaces shown between items are required unless one of the items is a punctuation symbol such as a parenthesis or a comma. For example:

```
CALL STEPMOM ( process-id ) ;
```

If there is no space between two items, spaces are not permitted. In this example, no spaces are permitted between the period and any other items:

```
$process-name.#su-name
```

Line Spacing. If the syntax of a command is too long to fit on a single line, each continuation line is indented three spaces and is separated from the preceding line by a blank line. This spacing distinguishes items in a continuation line from items in a vertical list of selections. For example:

```
ALTER [ / OUT file-spec / ] LINE

   [ , attribute-spec ]...
```

!i and !o. In procedure calls, the !i notation follows an input parameter (one that passes data to the called procedure); the !o notation follows an output parameter (one that returns data to the calling program). For example:

```
CALL CHECKRESIZESEGMENT (  segment-id                    !i
                         , error        ) ;              !o
```

!i,o. In procedure calls, the !i,o notation follows an input/output parameter (one that both passes data to the called procedure and returns data to the calling program). For example:

```
error := COMPRESSEDIT ( filenum ) ;                    !i,o
```

!i:i. In procedure calls, the !i:i notation follows an input string parameter that has a corresponding parameter specifying the length of the string in bytes. For example:

```
error := FILENAME_COMPARE_ (  filename1:length         !i:i
                            , filename2:length ) ;     !i:i
```

!o:i. In procedure calls, the !o:i notation follows an output buffer parameter that has a corresponding input parameter specifying the maximum length of the output buffer in bytes. For example:

```
error := FILE_GETINFO_ (  filenum                      !i
                        , [ filename:maxlen ] ) ;      !o:i
```

Notation for Messages

This list summarizes the notation conventions for the presentation of displayed messages in this manual.

Bold Text. Bold text in an example indicates user input typed at the terminal. For example:

```
ENTER RUN CODE
?123
CODE RECEIVED:      123.00
```

The user must press the Return key after typing the input.

Nonitalic text. Nonitalic letters, numbers, and punctuation indicate text that is displayed or returned exactly as shown. For example:

```
Backup Up.
```

lowercase italic letters. Lowercase italic letters indicate variable items whose values are displayed or returned. For example:

p-register

process-name

[ ] Brackets. Brackets enclose items that are sometimes, but not always, displayed. For example:

Event number = *number* [Subject = *first-subject-value*]

A group of items enclosed in brackets is a list of all possible items that can be displayed, of which one or none might actually be displayed. The items in the list can be arranged either vertically, with aligned brackets on each side of the list, or

horizontally, enclosed in a pair of brackets and separated by vertical lines. For example:

```
proc-name trapped [ in SQL | in SQL file system ]
```

{ } Braces. A group of items enclosed in braces is a list of all possible items that can be displayed, of which one is actually displayed. The items in the list can be arranged either vertically, with aligned braces on each side of the list, or horizontally, enclosed in a pair of braces and separated by vertical lines. For example:

```
obj-type obj-name state changed to state, caused by
{ Object | Operator | Service }
```

```
process-name State changed from old-objstate to objstate
{ Operator Request. }
{ Unknown.          }
```

| Vertical Line. A vertical line separates alternatives in a horizontal list that is enclosed in brackets or braces. For example:

```
Transfer status: { OK | Failed }
```

% Percent Sign. A percent sign precedes a number that is not in decimal notation. The % notation precedes an octal number. The %B notation precedes a binary number. The %H notation precedes a hexadecimal number. For example:

```
%005400
```

```
%B101111
```

```
%H2F
```

```
P=%p-register E=%e-register
```

Notation for Management Programming Interfaces

This list summarizes the notation conventions used in the boxed descriptions of programmatic commands, event messages, and error lists in this manual.

UPPERCASE LETTERS. Uppercase letters indicate names from definition files. Type these names exactly as shown. For example:

```
ZCOM-TKN-SUBJ-SERV
```

lowercase letters. Words in lowercase letters are words that are part of the notation, including Data Definition Language (DDL) keywords. For example:

```
token-type
```

!r. The !r notation following a token or field name indicates that the token or field is required. For example:

```
ZCOM-TKN-OBJNAME      token-type ZSPI-TYP-STRING.          !r
```

!o. The !o notation following a token or field name indicates that the token or field is optional. For example:

```
ZSPI-TKN-MANAGER      token-type ZSPI-TYP-FNAME32.          !o
```

Change Bar Notation

Change bars are used to indicate substantive differences between this manual and its preceding version. Change bars are vertical rules placed in the right margin of changed portions of text, figures, tables, examples, and so on. Change bars highlight new or revised information. For example:

> The message types specified in the REPORT clause are different in the COBOL environment and the Common Run-Time Environment (CRE).
>
> The CRE has many new message types and some new message type codes for old message types. In the CRE, the message type SYSTEM includes all messages except LOGICAL-CLOSE and LOGICAL-OPEN.

HP Encourages Your Comments

HP encourages your comments concerning this document. We are committed to providing documentation that meets your needs. Send any errors found, suggestions for improvement, or compliments to docsfeedback@hp.com.

Include the document title, part number, and any comment, error found, or suggestion for improvement that you have concerning this document.

1

Introduction to Guardian Procedure Calls

System services are tasks such as retrieving a record from a disk, writing a file to a tape, sending messages to other processes, or alerting your process to some kind of system error that the operating system or a subsystem performs on behalf of a program.

Your programs can make use of these services by calling appropriate Guardian procedures. For example, using the Guardian READ procedure allows a program to read data from a file.

To help you understand how to use the Guardian procedure-call descriptions in this manual, this section describes:

- The different types of Guardian procedure calls
- Reference parameter overlap
- H-series and G-series Guardian procedures and superseded Guardian procedures
- External declaration files for Guardian procedures
- Parameter declaration files for Guardian procedures
- An example of a Guardian procedure call
- Calling Guardian procedures from a Transaction Application Language (TAL) application
- Calling Guardian procedures from a C application

This manual describes most of the Guardian procedures and shows the syntax for calling these procedures from a TAL or C program.

You can also call Guardian procedures from programs written in FORTRAN, BASIC, COBOL, or C++. How a Guardian procedure is called depends on the programming language. Some languages provide extensions for calling Guardian procedures. Some languages (other than TAL) allow your code to contain TAL code that calls Guardian procedures using TAL syntax. For information on calling the Guardian procedures described in this manual from other languages, see the appropriate manual:

For information on translating calls in this manual to:	See:
FORTRAN	*FORTRAN Reference Manual*
BASIC	*EXTENDED BASIC Programmer's Guide*
COBOL	*COBOL85 for NonStop Systems Manual* *COBOL Manual for TNS/E Programs*
C	*C/C++ Programmer's Guide*
C++	*C/C++ Programmer's Guide*

For applications written in programming languages other than TAL, be cautious about using TAL to call Guardian procedures. This method of invoking the services provided by Guardian procedures can interfere with the run-time environment established for the programming language. If possible, use an extension of your programming language instead of embedded TAL code to invoke a Guardian service (such as reading from or writing to a file).

Types of Guardian Procedure Calls

Table 1-1 shows the types of Guardian procedures that are described in this manual. It also shows the manuals where you can find programming information about these procedures. For a complete list of all documented Guardian procedures and the manuals in which they are described, see Appendix H, Documented Guardian Procedures.

Table 1-1. Types of Guardian Procedure Calls (page 1 of 2)

Procedure Type	Action	Programming Manuals
DEFINEs	Specify DEFINEs by class and attribute values.	*Guardian Programmer's Guide*
File system	Perform operations, such as input and output, on files (this set of procedures includes Enscribe procedures).	*Guardian Programmer's Guide* *Enscribe Programmer's Guide*
Formatter	Format output data and convert input data.	*Guardian Programmer's Guide*
Memory management	Allocate extended memory segments and pools; provide exclusive access to data.	*Guardian Programmer's Guide*

Table 1-1. Types of Guardian Procedure Calls (page 2 of 2)

Procedure Type	Action	Programming Manuals
Process control	Run, suspend, activate, and stop programs.	*Guardian Programmer's Guide*
Security	Control access to processes and disk files.	*Guardian Programmer's Guide* *Safeguard Reference Manual*
Sequential I/O (SIO)	Perform sequential input and output operations to files.	*Guardian Programmer's Guide*
Signal manipulation	Detect critical error conditions in a native process.	*Guardian Programmer's Guide* *Open System Services Programmer's Guide*
Trap and trap handling	Detect critical error conditions in a TNS process.	*Guardian Programmer's Guide*
Utility	Perform miscellaneous operations such as translating a number from displayed (string) form to integer form and vice versa or getting a timestamp.	*Guardian Programmer's Guide*

The *Guardian Programmer's Guide* describes how to use many of these Guardian procedures according to their function and type (the file system, formatter, memory management, process control, security, SIO, signal manipulation, trap and trap handling, and utility procedures). However, the *Guardian Procedure Calls Reference Manual* presents Guardian procedure descriptions in alphabetical order.

This manual provides this information for each Guardian procedure:

- Syntax
- Parameter descriptions
- Condition codes
- Considerations
- Examples
- Manual references

H-Series Guardian Procedures

The H-series Guardian procedures support many of the G-series operating system features and provide further process control functionality. For details about how to convert applications to use the H-series Guardian procedures, see the *H-Series*

Application Migration Guide. For further information about how to use the operating system features, see the *Guardian Programmer's Guide*.

Superseded Guardian procedures continue to be supported for compatibility and are still documented in this manual. They are marked as "superseded" and are listed in Appendix G, Superseded Guardian Procedure Calls and Their Replacements.

G-Series Guardian Procedures

The G-series Guardian procedures include the features of the D-series operating system. Some features on the G-series RVUs are not available on D-series RVUs. For details on how to convert applications to use G-series Guardian procedures, see the *G-Series System Migration Planning Guide*. For details on how to use the features of the operating system, see the *Guardian Programmer's Guide*.

Superseded Guardian procedures continue to be supported for compatibility and are still documented in this manual. They are marked as "superseded" and are listed in Appendix G, Superseded Guardian Procedure Calls and Their Replacements.

For G-series and later Guardian applications, the variable supplied as an output parameter in a call to a Guardian procedure cannot be the same as, nor overlap, any of the other reference parameters to the routine.

Interprocess communication is not supported when G-series nodes and C-series nodes both exist in a network.

External Declarations Files for Guardian Procedures

Like all procedures in an application program, Guardian procedures must be declared before they can be called. Guardian procedures are declared as external procedures. A $SYSTEM.SYSTEM file contains many of the Guardian procedure declarations for each programming language. For example:

- pTAL declarations are in $SYSTEM.SYSTEM.EXTDECS0.
- C declarations are in $SYSTEM.SYSTEM.CEXTDECS (usually referred to as the `cextdecs` file).

Other header files in $SYSTEM.SYSTEM also contain Guardian procedure declarations. For example:

- $SYSTEM.ZGUARD.HSETJMP contains declarations for procedures that perform nonlocal goto operations.
- $SYSTEM.SYSTEM.HTDMSIG contains declarations for the signal-handling procedures.

The `setjmp.h` and `tdmsig.h` header files contain the equivalent C language declarations.

Your application should include the appropriate compiler instructions specifying the names of the appropriate external declarations files and the Guardian procedures that your application calls. The compiler instruction must precede the first invocation of a Guardian procedure. For information about compiler instructions, see your programming language reference manual. For example:

- See the *pTAL Reference Manual* for details on using the SOURCE compiler command with the TAL and pTAL programming languages.
- See the *C/C++ Programmer's Guide* for details on using the `#include` compiler command with the C programming language.

The procedure syntax descriptions for TAL and pTAL throughout this manual assume that an external declarations file (EXTDECS0, EXTDECS1, or EXTDECS) on $SYSTEM.SYSTEM is declared. Where a different header file is needed, the appropriate SOURCE compiler command is shown with the syntax.

Multiple versions of the external declarations file are provided for compiling your program to run on previous versions of the operating system as well as on the current version; older versions, however, do not support pTAL. Procedures that require EXTDECS0 are noted. For further information, see the *Guardian Programmer's Guide*.

Parameter Declarations Files for Guardian Procedures

HP provides a set of declarations, consisting mainly of named constants (literals) and structure definitions, that can be used for parameters to Guardian procedures. Data Definition Language (DDL) is used to generate files containing the parameter declarations for TAL, COBOL, Pascal, and C.

These files are located on the subvolume $SYSTEM.ZSYSDEFS. The files ZSYSTAL, ZSYSCOB, ZSYSPAS, and ZSYSC contain declarations for TAL, COBOL, Pascal, and C, respectively. The DDL declarations that are used to generate the ZSYS files are in the file ZSYSDDL.

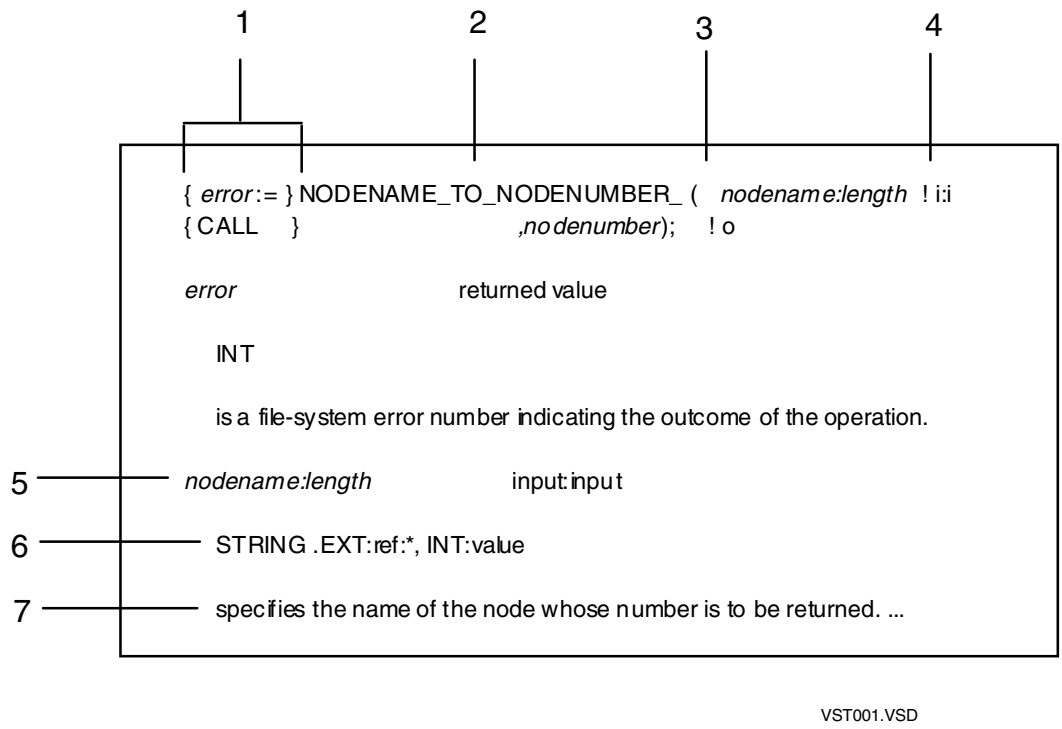
Your application should include the appropriate compiler instruction specifying the name of the parameter declarations file you want to use in your application. For information about compiler instructions, see your programming language reference manual.

For example, to use the TAL parameter declarations in a TAL program, you would include a SOURCE compiler command specifying the file $SYSTEM.ZSYSDEFS.ZSYSTAL. The SOURCE command should follow the global declarations and precede the first use of an item from ZSYSTAL in your source program. For details on using the SOURCE command in a TAL program, see the *TAL Reference Manual*.

TAL Syntax for a Guardian Procedure Call

An example of the TAL syntax description used in this manual is shown in Figure 1-1.

Figure 1-1. Sample TAL Syntax for a Procedure Call



The numbered items in the diagram are described below:

1 This indicates that the procedure is a function procedure; it returns a value of the indicated type (in this case INT) when referenced in an expression. You can specify the variable as *retval*, *status*, *error^code*, or some other appropriate name in other function procedure calls.

For function procedures that set the condition code, you must declare the return variable as a simple variable. If you declare it as a subscripted variable or a structure element, assigning a value to it can alter the condition code.

"CALL" is a TAL CALL statement. Any procedure that does not return a value must be invoked by a TAL CALL statement. In addition, you can use a CALL statement to invoke a function procedure if you do not need the returned value.

2 This is the name of the procedure that is called. The name is not case-sensitive but otherwise must appear in the program exactly as shown.

3 You must enclose the list of parameters in parentheses. Use commas to separate parameters when there is more than one. If you omit optional parameters, a placeholder comma (,) must be present for each omitted parameter unless you omit it from the end of the list. An optional parameter is indicated in a syntax diagram by a parameter name enclosed in square brackets.

Two parameters separated by a colon are treated as a unit. If they are optional parameters, both members of the pair must be either present or absent. If you omit the pair within a list of parameters, use only a single placeholder comma. In references to the ordinal numbers of parameters, the pair is considered one parameter.

4 The exclamation point indicates that a comment follows. The comment can be an "i", an "o", or "i,o", indicating that the parameter is either an input (i) parameter, an output (o) parameter, or both. In the example shown, "i:i" indicates that *nodename* and *length* are both input parameters.

5 This line also indicates whether the parameter is an input parameter, an output parameter, or both. In the example shown, "input:input" indicates that *nodename* and *length* are both input parameters.

6 This line indicates the parameter type. In the example shown, the types for *nodename* and *length* are both given, separated by a comma. The possible parameter types include:

INT	16-bit integer
INT(32)	32-bit integer
STRING	character string (8-bit character)
FIXED	64-bit fixed-point number
REAL	32-bit floating-point number
EXTADDR	32-bit address

For a complete discussion of formal parameter specifications, see the *pTAL Reference Manual*.

The parameter type is followed by a colon. Additional information after the colon includes:

value	means the actual value or contents of a parameter are passed.
ref:x	means that this is a reference parameter, that is, the address of the parameter is passed. (The statements within the program must access the actual parameter contents indirectly through the parameter location.) *x* indicates the number of elements the parameter contains. In this example, * indicates that the number of elements in the *nodename* parameter depends on another variable (in this case, *length*).
.EXT	means the parameter is a reference parameter accessed by an extended pointer.

If a parameter is defined as “STRING:ref,” or “STRING .EXT:ref,” an integer variable can be passed for “STRING:ref” parameter in TAL, the compiler produces instructions to convert address of the integer variable to the address (as a number of 8-bit characters) of the first character (byte) of that integer variable; this conversion is erroneous if the original address is greater than 32767.

For “STRING:ref,” parameters in TAL or for pTAL, no address conversion is necessary and the limit does not apply.

7 This describes the information that is passed or returned in the parameter.

String Output Variables

The syntax for some Guardian procedures contains one or more sets of three parameters that are grouped together, where each set describes a string output variable. Figure 1-2 shows an example of this use.

Note that the first two parameters are separated by a colon. (See Figure 1-1 on page 1-6 for a general description of the use of two parameters separated by a colon.) The *filename* parameter is an output parameter that contains a character string on return; *maxlen* is an input parameter that specifies the maximum number of characters that can be returned in *filename*; *filename-length* is an output parameter that returns the actual number of characters returned in *filename*.

When three parameters are grouped in this fashion, all of them must be either present or absent. If only one or two of them are present, an error is returned.

Figure 1-2. Syntax With String Output Variable

```
error := PROCESSHANDLE_TO_FILENAME_ ( processhandle     ! i
                                    ,filename:maxlen    ! o:i
                                    ,filename-length    ! o
                                         . . .
```

Reference Parameter Overlap

No variable that you supply as an output parameter in a call to a Guardian procedure should have the same address as, or overlap, any other reference parameter to the procedure. The only exceptions to this occur where the procedure description explicitly allows such use.

Bounds Checking of Reference Parameters for Guardian Procedures

Starting in the D20 RVU, bounds checking of reference parameters to Guardian procedures is different. The change does not affect programs that call procedures with correct parameters that are within bounds; these programs will continue to work correctly.

In some cases where reference parameters to Guardian procedures point to areas that were formerly considered out of bounds, the procedure might not detect the error. If the out-of-bounds parameter would either breach security or compromise system integrity, however, it will continue to be detected.

As an example, many Guardian procedures now use a local data stack that is separate from the user's local data stack. Before D20, it was an error to specify a reference to an array that extended beyond the end of the user's local data area into the data area of the called Guardian procedure. Starting in D20, you cannot rely on a Guardian procedure to return an error when you specify a reference parameter in this manner.

Those Guardian procedures that use a different local data stack copy the caller's parameters to that stack before proceeding. Because the reference parameter does not then intrude into the Guardian procedure's local data stack, it does not cause a bounds violation error. It is possible, however, that the process will abnormally terminate on return from the called Guardian procedure, because it might have written over the return address that was stored in the local stack.

Which Guardian procedures actually switch stack might change from RVU to RVU and will differ by processor type.

C Syntax for a Guardian Procedure Call

C syntax is presented in this manual in addition to TAL syntax. Where necessary, considerations for C programmers are also presented. For further information on calling Guardian procedures from a C program, see the *C/C++ Programmer's Guide*.

Figure 1-3 shows an example of C syntax. As in TAL syntax, square brackets ([ ]) indicate optional parameters. Detailed descriptions of parameters are not included with the C syntax; such descriptions accompany the TAL syntax.

Figure 1-3. Sample C Syntax for a Procedure Call

```
#include <cextdecs(BREAKMESSAGE_SEND_)>

short BREAKMESSAGE_SEND_ ( short *processhandle
                          ,short receiver-filenum
                          ,[ short *breaktag ] );
```

Because TAL does not have a string terminator like C, it often requires that you supply both a string and its length (not counting the null-byte terminator) in two parameters, or that you supply a string buffer with the maximum string length along with a third parameter for returning the actual string length (see String Output Variables on page 1-9).

When calling TAL procedures that use strings in this manner from a C program, you must also pass the complete set of parameters for handling the string. The TAL convention of pairing certain string parameters together, joined by colons, is not supported in C; parameters are always separated by commas.

C Header Files

To support portability from the UNIX environment, the name of a C header file can contain an internal period (.). The HP C for NonStop Systems compiler accepts both of these statements as equivalent:

```
#include <stdio.h>
#include <stdioh>
```

Actual C header files are stored on the same subvolume as the C compiler with no internal periods in their names. If the C compiler on your system is

$SYSTEM.SYSTEM.C, then your system should have the header file $SYSTEM.SYSTEM.STDIOH.

CEXTDECS in H-Series Systems

In H-series systems, CEXTDECS (through the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

How to find the (writable) global data in an TNS/E native process

The global, writable data in a native TNS/E process can be found in up to four distinct sections per loadfile[1]: data, data1, sdata, sdata1, sbss, and bss. The first two are for initialized data and the last two are for uninitialized data (which is always set to zero by NSK). Some (or even all) of them could be zero length. They may not be contiguous.

The base addresses and lengths of each section can be determined from linker-defined reserved symbols.

The simplest way to get this information is by coding a small C function that accesses the symbols and returns the addresses and lengths. That function can be compiled in C and the resulting object file linked into an otherwise completely pTAL program, without requiring the use of additional runtime support (no CRTL and CRE DLLs are required). Note that to report about the program, the function must be linked into the program loadfile; if it were in a separate DLL it would report about the instance data sections of that DLL, since these special symbols are local to the each loadfile.

1. Normally programs that use passive checkpointing (that is, use the file system CHECKPOINT functions) consist of a single loadfile: the program file with no DLLs. If there is a DLL (perhaps a UL), a similar function with another name can be linked into it to report the DLL's instance data segments.

Examples

- This example is of a function that returns the addresses and their lengths. Note that if the length is 0, the address is not significant.

```
typedef struct  {
  void * baseAddress;
  intlength;
} globalLocation[4];

extern char _data_start;
extern char _data_end;
extern char _sdata_start;
extern char _sdata_end;
extern char _sbss_start;
extern char _sbss_end;
extern char _bss_start;
extern char _bss_end;

void GETGLOB (globalLocation g) {
  g[0].baseAddress = &_data_start;
  g[0].length = &_data_end-&_data_start;
  g[1].baseAddress = &_sdata_start;
  g[1].length = &_sdata_end-&_sdata_start;
  g[2].baseAddress = &_bss_start;
  g[2].length = &_bss_end-&_bss_start;
  g[3].baseAddress = &_sbss_start;
  g[3].length = &_sbss_end-&_sbss_start;
}
```

- This is an example of an EPTAL program skeleton that uses the previous function:

```
struct globalLocation[0:3];
begin
  extaddr baseAddress;
  int(32) length;
end;

proc GETGLOB(g);
  int .ext g(globalLocation);
  external;

?source $system.system.extdecs(DEBUG)

proc m main;
begin
  GETGLOB(globalLocation);
  DEBUG;
end;
```

- This is an example of the contents of the global Location from eInspect at that DEBUG call:

```
(eInspect 1,455):p GLOBALLOCATION
$1 = {{
    BASEADDRESS = 0x8000000,
    LENGTH = 0
  }, {
    BASEADDRESS = 0x8000090,
    LENGTH = 0
  }, {
    BASEADDRESS = 0x8000090,
    LENGTH = 0
  }, {
    BASEADDRESS = 0x8000090,
    LENGTH = 48
  }}
```

2

Guardian Procedure Calls (A-B)

This section contains detailed reference information for all user-accessible Guardian procedure calls beginning with the letters A through B. Table 2-1 lists all the procedures in this section.

Table 2-1. Procedures Beginning With the Letters A Through B

ABEND Procedure (Superseded by PROCESS_STOP_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The ABEND procedure deletes a process or process pair and signals that the deletion was caused by an abnormal condition. When this procedure is used to delete a Guardian process or an Open System Services (OSS) process, an ABEND system message is sent to the deleted process's creator. When this procedure is used to delete an OSS process, a `SIGCHLD` signal and the OSS process termination status are sent to the OSS parent process.

A process can use ABEND to:

- Delete itself
- Delete its own backup
- Delete another process

When the ABEND procedure is used to delete a Guardian process, the caller must either have the same process access ID as the process it is attempting to abend, be the group manager of the process access ID (255,255), or be the super ID. For the PROCESSACCESSID procedure, see Considerations on page 2-5 and for a description of the process access ID, see *Guardian Programmer's Guide*.

When ABEND is used on an OSS process, the same security rules apply as for the OSS `kill()` function.

When ABEND executes, all open files associated with the deleted process are automatically closed. If a process had BREAK enabled, BREAK is disabled.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with

previous software.

Syntax for TAL Programmers

```
CALL ABEND ( [ process-id ]                    ! i
           ,[ stop-backup ]                    ! i
           ,[ error ]                          ! o
           ,[ compl-code ]                     ! i
           ,[ termination-info ]               ! i
           ,[ spi-ssid ]                       ! i
           ,[ length ]                         ! i
           ,[ text ] );                        ! i
```

Parameters

process-id input

INT:ref:4

indicates the process that is to be stopped. At this point, you have two options.

The value you enter can be either:

- Omitted (or zero), meaning “stop myself”, or
- The 4-word array containing the process ID of the process to be stopped, where:

[0:2]		Process name or creation timestamp
[3]	.<0:3>	Reserved
	.<4:7>	Processor number where the process is executing
	.<8:15>	PIN assigned by the operating system to identify the process in the processor

If *process-id*[0:2] references a process pair and *process-id*[3] is specified as -1, then both members of the process pair are stopped.

stop-backup input

INT:value

if specified as 1, the current process’s backup is stopped and ABEND is returned to the caller. The *process-id* parameter is not used.

If zero, this parameter is ignored and the *process-id* parameter is used as described.

error output

INT:ref:1

returns a file-system error number. ABEND returns a nonzero value for this parameter only when it cannot successfully make the request to stop the

designated process. If it makes the request successfully (*error* is 0), the designated process might or might not be stopped depending on the stopmode of the process and the authority of the caller. (The stop mode of the process can be changed; hence, a stop request that has inadequate authority to stop the process is saved by the system and might succeed at a later time.) See Considerations on page 2-5.

These parameters supply completion-code information, which consists of four items: the completion code, a numeric field for additional termination information, a subsystem identifier in SPI format, and an ASCII text string. These items have meaning in the call to ABEND only when a process is stopping itself.

compl-code input

INT:value

is the completion code to be returned to the creator process in the ABEND system message and, for a terminating OSS process, in the OSS termination status. Specify this parameter only if the calling process is terminating itself and you want to return a completion code value other than the default value of 5. For a list of completion codes, see Appendix C, Completion Codes.

termination-info input

INT:value

can be provided as an option by the calling process if it is a subsystem process that defines Subsystem Programmatic Interface (SPI) error numbers. If supplied, this parameter should be the SPI error number that identifies the error that caused the process to stop itself. For more information on the SPI error numbers and subsystem IDs, see the *SPI Programming Manual*. If *termination-info* is not specified, this field is zero.

spi-ssid input

INT .EXT:ref:6

is a subsystem ID (SSID) that identifies the subsystem defining *termination-info*. The format and use of the SSID is described in the *SPI Programming Manual*.

length input

INT:value

is the length in bytes of *text*. The maximum length is 80 bytes.

text input

STRING .EXT:ref:*

is an optional string of ASCII text to be sent in the ABEND system message.

Condition Code Settings

A condition code value is returned only when a process is calling ABEND on another process and that other process could not be terminated.

< (CCL) indicates that either the *process-id* parameter is invalid or an error occurred during termination of the process.

= (CCE) indicates that ABEND was successful.

> (CCG) is not returned from ABEND.

Considerations

- Differences between ABEND and STOP procedures

  When used to stop the calling process, the ABEND and STOP procedures operate almost identically; they differ in the system messages that are sent and the default completion codes that are reported. In addition, ABEND, but not STOP, causes a saveabend file to be created if the process's SAVEABEND attribute is set to ON. See the *Inspect Manual* for information about saveabend files.

- Creator of the process and the caller of ABEND

  If the caller of ABEND is also the creator of the process being deleted, the caller receives the ABEND system message.

- Rules for stopping a Guardian process: process access IDs and creator access IDs

  If the process is a local process and the request to stop it is also from a local process, these user IDs or associated processes may stop the process:

  - Local super ID (255, 255)
  - The process's creator access ID (CAID) or the group manager of the CAID
  - The process's process access ID (PAID) or the group manager of the PAID

  If the process is a local process, a remote process cannot stop it.

  If the process is a remote process running on this node and the request to stop it is from a local process on this node, these user IDs or associated processes may stop the process:

  - Local super ID
  - The process's creator access ID (CAID) or the group manager of the CAID
  - The process's process access ID (PAID) or the group manager of the PAID

  If the process is a remote process on this node and the request to stop it is from a remote process, these user IDs or associated processes can stop the process:

  - A network super ID

- The process's network process access ID
- The process's network process access ID group manager
- The process's network creator access ID
- The process's network creator access ID group manager

where network ID implies that the user IDs or associated process creators have matching remote passwords.

Being local on a system means that the process has logged on by successfully calling USER_AUTHENTICATE_ or VERIFYUSER on the system or that the process was created by a process that had done so. A process is also considered local if it is run from a program file that has the PROGID attribute set.

- Rules for stopping an OSS process

  The same rules apply when stopping an OSS process with the ABEND procedure as apply for the OSS `kill()` function. See the `kill(2)` function reference page either online or in the *Open System Services System Calls Reference Manual*.

- Rules for stopping any process: stopmode

  When one process attempts to stop another process, another item checked is the "stopmode" of the process. Stopmode is a value associated with every process that determines which other processes can stop the process. The stopmode, set by the SETSTOP procedure, is defined below:

  0 ANY other process can stop the process;
  1 ONLY the process qualified by the above rules can stop the process;
  2 NO other process can stop the process.

- Returning control to the caller before the process is stopped

  When *error* is 0, ABEND returns control to the caller before the specified process is actually stopped. Although the process does not execute any more user code, you should make sure that it has terminated before you attempt to access a file that it had open with exclusive access or before you try to create a new process with the same name. The best way to be sure that a process has terminated is to wait for the process deletion message.

- Stopping a process that has the Inspect or saveabend attribute set

  If the process being stopped has either the Inspect attribute or the saveabend attribute set, and if DMON exists, ABEND returns error 0 but deletion of the process is delayed until DMON approves it. If the saveabend attribute is set, DMON creates a saveabend file.

- In response to the ABEND procedure, the operating system supplies a completion code in the system message and, for OSS processes, in the OSS process termination status as follows:

  - If a process calls ABEND on another process, the system supplies a completion code value of 6.

- If a process calls ABEND on itself but does not supply a completion code, the system supplies a completion code value of 5.

  For a list of the completion codes, see Appendix C, Completion Codes.

- Deleting high-PIN processes

  ABEND cannot be used to delete a high-PIN unnamed process, but it can use it to delete a high-PIN named process or process pair.

  A high-PIN caller (named or unnamed) can delete itself by omitting *process-id*.

NetBatch Considerations

- The ABEND procedure supports NetBatch by:
  - Returning the completion code information in the ABEND system message.
  - Returning the process processor time in the ABEND system message
  - Sending an ABEND system message to the ancestor of a job (the GMOM) as well as the ancestor of a process

Messages

- Process deletion (ABEND) message

  The creator of the stopped process is sent a system message -6 (process deletion: ABEND) indicating that the deletion occurred. For the format of the interprocess system messages, see the *Guardian Procedure Errors and Messages Manual*.

OSS Considerations

- When an OSS process is stopped by the ABEND procedure, either by calling the procedure to stop itself or when some other process calls the procedure, the OSS parent process receives a `SIGCHLD` signal and the OSS process termination status. For details on the OSS process termination status, see the `wait(2)` function reference page either online or in the *Open System Services System Calls Reference Manual*.

  In addition, an ABEND system message is sent to the MOM, GMOM, or ancestor process according to the usual Guardian rules.

- When the ABEND procedure is used to terminate an OSS process other than the caller, the Guardian process ID must be specified in the call. The effect is the same as if the OSS `kill()` function was called with the input parameters as follows:
  - The *signal* parameter set to `SIGABEND`
  - The *pid* parameter set to the OSS process ID of the process identified by *processhandle* in the PROCESS_STOP_ call

- The security rules that apply to terminating an OSS process using ABEND are the same as those that apply to the OSS `kill()` function. For details, see the `kill(2)` function reference pages either online or in the *Open System Services System Calls Reference Manual*.

Examples

```
CALL ABEND;                 ! cause this process to abend.
CALL ABEND ( ProcID );      ! cause the process that has
                            ! this process ID to abend.
```

Related Programming Manual

For information on batch processing, see the *NetBatch User's Guide*.

ACTIVATEPROCESS Procedure (Superseded by PROCESS_ACTIVATE_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The ACTIVATEPROCESS procedure returns a process or process pair from the suspended state to the ready state. (A process is put in the suspended state if it is the object of a call to the SUSPENDPROCESS procedure, or if it is suspended as the result of a SUSPEND command issued from the command interpreter.)

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
CALL ACTIVATEPROCESS ( process-id );              ! i
```

Parameters

process-id input

INT:ref:4

is a 4-word array containing the process ID of the process to be activated, where:

[0:2]	Process name or creation timestamp
[3].<0:3>	Reserved
.<4:7>	Processor number where the process is executing
.<8:15>	PIN assigned by the operating system to identify the process in the processor

If *process-id*[0:2] references a process pair and *process-id*[3] is specified as -1, then both members of the process pair are activated.

Condition Code Settings

< (CCL)	indicates that either ACTIVATEPROCESS failed or no process designated as *process-id* exists.
= (CCE)	indicates that the process is activated.
> (CCG)	is not returned from ACTIVATEPROCESS.

Considerations

- When ACTIVATEPROCESS is called on a Guardian process, the caller must be the super ID (255,255), the group manager ((n,255) of the process access ID, or a process with the same process access ID as the process or process pair being activated. For the PROCESSACCESSID procedure, see Considerations on page 2-5 and for information about the process access ID, see the *Guardian User's Guide*.
- When ACTIVATEPROCESS is called on an OSS process, the security rules that apply are the same as those that apply when calling the OSS `kill()` function. For details, see the `kill(2)` function reference page either online or in the *Open System Services System Calls Reference Manual*.
- ACTIVATEPROCESS cannot be used on a high-PIN unnamed process. However, it can be used on a high-PIN *named* process or process pair; *process-id*[3] must contain either -1 or two blanks.

To activate a high-PIN unnamed process, use the PROCESS_ACTIVATE_ procedure. See the *Guardian Programmer's Guide*.

OSS Considerations

When used on an OSS process, ACTIVATEPROCESS has the same effect as calling the OSS `kill()` function with the input parameters as follows:

- The *`signal`* parameter set to `SIGCONT`
- The *`pid`* parameter set to the OSS process ID of the process identified by *`process-id`* in the ACTIVATEPROCESS call

The `SIGCONT` signal is delivered to the target process.

ADDDSTTRANSITION Procedure (Superseded by DST_GETINFO_ Procedure)

Summary

The ADDDSTTRANSITION procedure allows a super-group user (255,*n*) to add an entry to the daylight-saving-time (DST) transition table. This operation is allowed only when the DAYLIGHT_SAVING_TIME option in the system is configured to the TABLE option.

Syntax for C Programmers

```
#include <cextdecs(ADDDSTTRANSITION)>

_cc_status ADDDSTTRANSITION ( long long low-gmt
                             ,long long high-gmt
                             ,short offset );
```

- The function value returned by ADDDSTTRANSITION, which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL ADDDSTTRANSITION ( low-gmt                    ! i
                       ,high-gmt                   ! i
                       ,offset );                  ! i
```

Parameters

low-gmt input

FIXED:value

is the Greenwich mean time (GMT) when *offset* is first applicable. (This form is the same as the form used for COMPUTETIMESTAMP.) Except for the first call, the *low-gmt* parameter of each call must be the same as the *high-gmt* parameter of the previous call. This implies that many calls have an *offset* parameter of 0.

high-gmt input

FIXED:value

is the GMT when *offset* is no longer applicable.

offset input

INT:value

is this value in seconds:

local civil time (LCT) = local standard time (LST) + *offset*

Condition Code Settings

< (CCL) indicates that you:

- Are not a super-group user (255,*n*)

- Loaded the DST table inconsistently (that is, the DST table contains an overlap of entries)
- Were loading the DST table at the same time someone else was loading the DST table.

= (CCE) indicates that the DST table was loaded successfully.

> (CCG) is not returned from ADDDSTTRANSITION.

Considerations

- Application programs and utilities such as BACKUP cannot reference any date prior to the first entry in the DST transition table.

ADDRESS_DELIMIT_ Procedure

Summary

The ADDRESS_DELIMIT_ procedure obtains the addresses of the first and last bytes of a particular area of the caller's logical address space. It can also obtain a set of flags that describe the area, and the logical segment ID of the area.

Syntax for C Programmers

```
#include <cextdecs(ADDRESS_DELIMIT_)>

short ADDRESS_DELIMIT_ ( __int32_t address
                        ,__int32_t *low-address
                        ,__int32_t *high-address
                        ,short *address-descriptor
                        ,short *segment-id
                        ,short error-detail );
```

Syntax for TAL Programmers

```
error := ADDRESS_DELIMIT_ ( address                    ! i
                          ,[ low-address ]             ! o
                          ,[ high-address ]            ! o
                          ,[ address-descriptor ]      ! o
                          ,[ segment-id ]              ! o
                          ,[ error-detail ] );         ! o
```

Parameters

error returned value

INT

indicates the outcome of the operation. It returns one of these values:

0 No error; the requested values are returned.

2 Parameter error; *address* parameter was missing.

3 Bounds error; *error-detail* contains the number of the first parameter found to be in error, where 1 designates the first parameter on the left. This error is returned only to nonprivileged callers.

4 The *address* parameter is not mapped.

5 The *address* parameter is one of these:

- An invalid address (*address*.<0> = 1); an address in the priv stack will have *bit* 0 = 1
- A relative segment address that is contained within either system data space, current code space, or user code space (that is, within relative segment 1, 2, or 3 respectively)
- On a TNS system, an address that is greater than the maximum extended address (that is, greater than %777777777D)

address input

EXTADDR:value

is a relative address contained within the address area about which information is desired. Note that *address* is an address passed by value, not a pointer passed as a reference parameter.

low-address output

EXTADDR .EXT:ref:1

if the value of *error* is either 0 (no error) or 4 (*address* is not mapped), returns the address of the first byte in the area that contains *address*. If *error* is 4, *low-address* returns the address of the first byte in the unmapped area.

high-address output

EXTADDR .EXT:ref:1

if the value of *error* is either 0 (no error) or 4 (*address* is not mapped), returns the address of the last byte in the area that contains *address*. If *error* is 4, *high-address* returns the address of the last byte in the unmapped area.

address-descriptor output

INT .EXT:ref:1

returns a value that contains a set of bit fields describing the address area that contains *address*. For details, see Address-Descriptor Bit Fields on page 2-15. If *error* is 4 (*address* is not mapped), *address-descriptor* returns 0.

segment-id output

INT .EXT:ref:1

returns the logical segment ID of the address area that contains *address*. Either this is the segment ID assigned by the caller when the segment was allocated, or it is a reserved segment ID. For details, see Reserved Segment ID Values on page 2-15. If *error* is 4 (*address* is not mapped), *segment-id* returns -1.

error-detail output

INT .EXT:ref:1

returns additional error information when an *error* value of 3 (bounds error) is returned. For details, see *error*.

General Considerations

For any output parameter to this procedure, supplying the parameter with the pointer address set to %37777000000D (null address) is equivalent to not supplying the parameter.

Address-Descriptor Bit Fields

The meanings of the bit fields returned by the *address-descriptor* parameter are:

Bit field	Indicates that the segment is
<0:4>	Bits are reserved; 0 is returned.
<0:5>	Bits are reserved; 0 is returned.
<5>	Managed by the Kernel-Managed Swap Facility (KMSF).
<6>	An OSS shared memory segment.
<7>	An unaliased segment. An unaliased segment does not have a corresponding absolute segment address.
<8>	A flat segment.
<9>	Currently in-use selectable segment for the process.
<10>	Accessible only by privileged processes.
<11>	Shared by another process.
<12>	Cannot be deallocated.
<13>	Read-only.
<14>	Extensible.
<15>	Resident.

Reserved Segment ID Values

The reserved segment ID values that can be returned by the *segment-id* parameter lie in the range -109 through -2 (65427 through 65534). These values are used internally to identify various types of segments allocated by the operating system, such as process stacks, global data, various kinds of code, and certain special segments. For some kinds of segments (such as SRL or DLL code or instance data), multiple segments in the process can have the same ID. Segment ID assignments are subject to change from RVU to RVU; the individual values are not meaningful to typical callers of ADDRESS_DELIMIT_. Current definitions can be found in these T9050 header files:

- DMEMH, beginning after identifier LAST_VALID_SSEDS_ID and ending before identifier NULL_PST_SEGID.
- DMEM, beginning after identifier LAST^VALID^SSEDS^ID and ending before identifier NULL^PST^SEGID.

These two header files are distributed and installed in the ZGUARD subvolume.

Example

In this example, the address of a local variable contained in the user data area is passed to ADDRESS_DELIMIT_. The procedure returns the addresses of the first and last bytes of the user data area.

This example shows that the output addresses can be assigned either to a simple variable (LOW^ADDR) or to a pointer variable (HIGH^ADDR). After a successful call to ADDRESS_DELIMIT_, HIGH^ADDR designates the last byte of the user data area.

```
INT       LOCAL^VARIABLE;
INT(32)   LOW^ADDR;
STRING   .EXT HIGH^ADDR;
INT       ERROR,
          ERROR^DETAIL;
            .
            .
            .
ERROR := ADDRESS_DELIMIT_ ($XADR(LOCAL^VARIABLE),
                           LOW^ADDR,
                           @HIGH^ADDR,
                           ! address^descriptor ! ,
                           ! segment^ID ! ,
                           ERROR^DETAIL);

IF ERROR <> 0 THEN CALL ERROR^HANDLER;
```

Related Programming Manual

For programming information about the ADDRESS_DELIMIT_ procedure, see the *Guardian Programmer's Guide*.

ADDRTOPROCNAME Procedure

Summary

Note. This procedure can be used only with TNS code. A comparable service is provided for accelerated code and native code using the HIST_INIT_ procedure with the HO_Init_Address option.

ADDRTOPROCNAME accepts a P register value and stack marker ENV value and returns the associated symbolic procedure name and various optional items that describe the procedure in detail.

Syntax for C Programmers

```
#include <cextdecs(ADDRTOPROCNAME)>

short ADDRTOPROCNAME ( short p-reg
                      ,short stack-env
                      ,char *proc-name
                      ,short proc-name-size
                      ,short *proc-name-length
                      ,[short *base ]
                      ,[short *size ]
                      ,[short *entry ]
                      ,[short *attributes ]
                      ,[short pin ] );
```

Syntax for TAL Programmers

```
error := ADDRTOPROCNAME ( p-reg                 ! i
                         ,stack-env             ! i
                         ,proc-name             ! o
                         ,proc-name-size        ! i
                         ,proc-name-length      ! o
                         ,[ base ]              ! o
                         ,[ size ]              ! o
                         ,[ entry ]             ! o
                         ,[ attributes ]        ! o
                         ,[ pin ] );            ! i
```

Parameters

error returned value

INT

returns a file-system error code indicating the outcome of the call, as follows:

0 Successful call; the procedure name is deposited into *proc-name* for *proc-name-length* bytes.

11 A procedure name was not found. This value is returned if an I/O error occurs during the read of the associated object file or if *p-reg*, *stack-env*, and the optional *pin* are valid but do not indicate a code location associated with a procedure (for example, a location in the PEP or XEP).

22 One of the parameters specifies an address that is out of bounds.

23 The *p-reg*, *stack-env*, and optional *pin* parameters do not indicate a valid code location.

24 The *pin* parameter was supplied and the caller is not privileged.

29 A required parameter was not supplied.

122 The supplied value of *proc-name-size* is less than the length of the procedure name that is to be returned into *proc-name*. The procedure name (and any other requested output parameters) is returned in *proc-name*, though it is truncated to the value of *proc-name-size*.

p-reg input

INT:value

is the target procedure's P register setting.

stack-env input

INT:value

is the target procedure's code space identifier in the stack marker ENV register format. Only these fields in *stack-env* are significant:

<4> Library bit
<7> System code bit
<11:15> Space ID bits

proc-name output

STRING .EXT:ref

is an ASCII string into which is returned the symbolic procedure name corresponding to the code location specified by *p-reg*, *stack-env*, and the optional *pin*.

proc-name-size input

INT:value

is the size, in bytes, of the caller's *proc-name* buffer.

proc-name-length output

INT .EXT:ref:1

is the length, in bytes, of the procedure name string returned into *proc-name*.

base output

INT .EXT:ref:1

is the base word address (first word) of the procedure indicated by *proc-name*.

size output

INT .EXT:ref:1

is the size, in words, of the procedure indicated by *proc-name*.

entry output

INT .EXT:ref:1

is the entry-point word address of the procedure indicated by *proc-name*.

attributes output

INT .EXT:ref:1

is a word describing attributes of the procedure indicated by *proc-name*. The *attributes* word contains these fields:

<0>	Privileged bit
<1>	Callable bit
<2>	Resident bit
<3>	Interrupt bit
<4>	Entry point bit
<5>	Variable bit
<6>	Extensible bit
<7:15>	PEP number

pin input

INT:value

specifies that *p-reg* and *stack-env* see the process identified by *pin* rather than the calling process. The *pin* parameter can be supplied only by privileged callers.

Considerations

- The maximum value of *proc-name-length*, and hence the address space that must be available at the location given by *proc-name*, depends on the language

that was used to create the code to which *p-reg*, *stack-env*, and the optional *pin* refer.

- Read access to the associated object file is not required in order to obtain the requested output parameters associated with the given *p-reg*, *stack-env*, and optional *pin*.

Example

```
INT     STACK^ENV  = 'L' - 1;    ! calling procedure's stack
                                    ENV
INT     P^REG      = 'L' - 2;    ! calling procedure's P
                                    register
LITERAL PROC^NAME^SIZE = 80;
STRING  PROC^NAME                 ! returned ASCII procedure
                                    name
        [ 0:PROC^NAME^SIZE-1 ];
INT     LENGTH;                   ! length of returned proc
                                    name
INT     BASE;                     ! procedure base address
INT     OFFSET;                   ! word offset within
                                    procedure

IF (ERROR := ADDRTOPROCNAME ( P^REG, STACK^ENV, PROC^NAME,
                              PROC^NAME^SIZE, LENGTH,
                              BASE ) ) THEN
   ! an error occurred, ERROR has the error code
ELSE
   OFFSET := P^REG '-' BASE;
```

ALLOCATESEGMENT Procedure (Superseded by SEGMENT_ALLOCATE_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The ALLOCATESEGMENT procedure allocates a selectable extended data segment for use by the calling process. This procedure can create read/write segments or read-only segments.

The ALLOCATESEGMENT procedure can also be used to share selectable extended data segments or flat extended data segments allocated by other processes (subject to the normal security requirements). Although it is possible to share flat segments using the ALLOCATESEGMENT procedure, flat segments can be allocated only by using the SEGMENT_ALLOCATE_ procedure. SEGMENT_ALLOCATE_ can also allocate selectable segments.

For selectable extended data segments, the call to ALLOCATESEGMENT must be followed by a call to USESEGMENT to make the segment accessible. Although you can allocate multiple selectable extended data segments, you can access only one at a time.

For shared flat segments, the call to ALLOCATESEGMENT can be followed by a call to USESEGMENT, but calling USESEGMENT is unnecessary because all the flat segments allocated by a process are always accessible to the process.

Flat segments and selectable segments are supported on native processors that use D30 or later RVUs of the HP NonStop operating system. Selectable segments are supported on all systems.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
status := ALLOCATESEGMENT ( segment-id              ! i
                          ,[ segment-size ]         ! i
                          ,[ file-name ]            ! i,o
                          ,[ pin-and-flags ] );     ! i
```

Parameters

status returned value

INT

indicates the outcome of the call:

0 No error

1-999 File-system error related to the creation or open of the swap file (see *file-name* parameter).

-1 Invalid *segment-id*.

-2 Invalid *segment-size*.

-3 Bounds violation on *file-name*.

-4 Invalid combination of options.

-5 Unable to allocate segment space.

-6 Unable to allocate segment page table space.

-7 Security violation on attempt to share segment.

-8 The *pin* parameter does not exist.

-9 The *pin* parameter does not have the segment allocated.

-10 Trying to share segment with self.

-11 Requested segment is currently being resized (delay and try again), or the requested segment is a shared selectable segment but the allocated segment is a flat segment.

segment-id input

INT:value

is the number by which the process chooses to refer to the segment. Segment IDs are in these ranges:

0-1023 Can be specified by user processes.
Other IDs Are reserved for HP software.

No nonprivileged process can supply a segment ID greater than 2047.

segment-size input

INT(32):value

specifies the size in bytes of the segment to be allocated.

Flat segment size:

- For G04.00 and earlier G-series RVUs the value must be in the range 1 byte through 128 megabytes (134,217,728 bytes). A flat segment is allocated beginning on a 32-megabyte region boundary and is allocated from a total virtual space of 480 megabytes (15 regions * 32 megabytes/region).
- For G05.00 and later G-series RVUs, the flat segment size limit is 1120 megabytes. Also, the 32-megabyte region boundary does not apply for these RVUs.

For a selectable segment, the value must be in the range 1 byte through 127.5 megabytes (133,693,440 bytes).

The system might round the size up to the next *segment-size* increment, where the increment is both processor-dependent and subject to change. The only effect this has on the program is that an address reference that falls outside the specified segment size but within the actual size does not cause an invalid address reference (trap 0 for a Guardian TNS process, a SIGSEGV signal for an OSS or any native process), and a subsequent fetch might not retrieve the value previously stored.

For methods of sharing segments, see the *pin-and-flags* parameter.

Upon initial allocation of the segment:

- The *segment-size* parameter is required if the swap file does not exist.
- The *segment-size* parameter is optional if the swap file already exists. If the segment is a read-only segment, the default size is the end-of-file value of the swap file (EOF). If the segment is a read-write segment, the default segment size is the allocated size of the swap file.
- For a read-only segment, *segment-size* must not be greater than the end-of-file value of the file; otherwise, an error occurs. For a read-write segment, if *segment-size* is greater than the allocated size of the swap file, the system attempts to allocate additional space.

If a segment is being shared by the PIN method (see *pin-and-flags*), this rule applies to the sharers:

- The *segment-size* parameter must be omitted and the size of the segment is the same as that from the initial ALLOCATESEGMENT call.

If a segment is being shared by the file name method (see *pin-and-flags*), these rules apply to the sharers:

- The *segment-size* parameter is optional. If the segment is a read-only segment, the default segment size is the length of the file (EOF). If the segment is a read-write segment, the default segment size is the allocated size of the file.
- For a read-only segment, *segment-size* must not be greater than the end-of-file value of the file; otherwise, an error occurs. For a read-write segment, *segment-size* must not be greater than the segment size specified by the initial call to ALLOCATESEGMENT.

file-name input, output

INT:ref:12

if present, is the internal-format file name of a swap file to be associated with the segment. If the file exists, all data in the file is used as initial data for the segment. If the file does not exist, one is created. Remote file names and structured files are not accepted. If the process terminates without deallocating the segment, any data still in memory is written back out to the file. ALLOCATESEGMENT must be able to allocate a sufficient number of file extents to contain all memory in the segment.

The parameter can be a volume name with a blank subvolume and file; ALLOCATESEGMENT allocates a temporary swap file on the indicated volume.

If you do not specify *file-name* and if a segment is not being shared using the PIN method, ALLOCATESEGMENT uses the Kernel-Managed Swap Facility (KMSF) to allocate swap space. To share this segment, use the PIN method; you cannot use the file-name method.

Performance is increased by using KMSF. However, if you want to save the data in the segment after the process terminates, specify a permanent swap file name. KMSF swap files have the clear-on-purge attribute, which provides a level of security for swapped data.

For more information on KMSF, see the *Kernel-Managed Swap Facility (KMSF) Manual*.

If a segment is being shared using the file-name method, *file-name* must be supplied. If a segment is being shared using the PIN method, *file-name* must be omitted.

pin-and-flags input

INT:value

Defaults to %040000. Its values are:

<8:15> Optional PIN for segment sharing.

Requests allocation of a shared segment that is shared by the PIN method. This value specifies the process identification number (PIN) of the process that has previously allocated the segment and with which the caller wants to share the segment. This value is not used if bit 1 is set to 1 (see bit <1> later). A shared segment is an extended data segment that can be shared with other processes in the same processor.

<5:7> Not used; must be zero (0).

<4> If 1, requests allocation of an “extensible segment.” An extensible segment is an extended data segment for which the underlying swap file disk space is not allocated until needed. In this case, *segment-size* is taken as a maximum size and the underlying swap file is expanded dynamically as the user accesses various addresses within the extended data segment. When the user first accesses a portion of an extensible segment for which the corresponding swap file extent has not been allocated, the operating system allocates the extent. If this extent cannot be allocated, the user process terminates: a TNS Guardian process terminates with a “no memory available” trap (trap 12); an OSS or native process receives a SIGNOMEM signal.

<3> If 1, requests allocation of a “shared segment” that is shared by the file-name method. A shared segment is an extended data segment that can be shared with other processes in the same processor. The *file-name* parameter must be supplied when this type of shared segment is allocated. (It is with the *pin-and-flags* parameter that sharing is specified.) Processes sharing a segment through the file-name method can reference the address space by different *segment-id*s and may supply different values of *segment-size* to ALLOCATESEGMENT. The *segment-size* supplied by the first allocator of a particular shared segment (as identified by the swap file name) will limit the size of the segment for subsequent processes attempting to share that segment. All processes that share segments

with the file-name method must have bit 3 and bit 1 set to 1 (see Examples on page 2-26).

<2> If 1, requests allocation of a “read-only segment.” A read-only segment is an extended data segment that is initialized from a preexisting swap file and used only for read access. A read-only segment can be shared by either the PIN or file-name method. It can also be shared by file name between processes in different processors. Note that the *file-name* parameter must specify the name of an existing swap file that is not empty. If this bit is 1, bit <4> of *pin-and-flags* must be 0 (writeback-inhibit extensible segments are not allowed) and bit 1 must be set to 1.

<1> If 1, bits <8:15> are ignored.

If 0, designates that segment sharing is to be done by the PIN method. The process calling ALLOCATESEGMENT with bit 1 set to 1 shares the segment specified with the currently running process specified by the PIN in bits <8:15> of the *pin-and-flags* word. The segment specified by *segment-id* must have been previously allocated by the process specified in the *pin-and-flags* word. Processes sharing a segment by this method reference the segment by the same *segment-id*.

Examples of valid *pin-and-flags* word values are:

%000*nnn*	Allocate a shared segment, to be shared using the PIN method with the process identified by the PIN specified in *nnn*.
%040000	Standard call to allocate a segment (default values).
%044000	Allocate an extensible segment.
%050000	Allocate a segment to be shared by the file-name method.
%054000	Allocate an extensible segment to be shared by the file-name method.
%060000	Allocate a read-only segment.

Considerations

- Preventing automatic temporary file purge

  ALLOCATESEGMENT opens the swap file for read/write protected access. A process can prevent the automatic file purge of a temporary swap file by opening the file for read-only shared access before the segment is deallocated.

- Nonexisting temporary swap file

  If a shared segment is being allocated (*pin-and-flags* bits <2:3> not equal to 0) and only a volume name is supplied in the *file-name* parameter, then the complete file name of the temporary file created by ALLOCATESEGMENT is returned.

- Swap file extent allocation

  If a shared extensible segment is being created, then only one extent of the swap file is allocated when ALLOCATESEGMENT returns. If a nonsharable extensible segment is being created, no extents are allocated until the user accesses the segment.

  Note that if ALLOCATESEGMENT creates the swap file, it configures the extent sizes based on a maximum of 64 extents.

- Segment sharing

  Subject to security requirements, a process can share a segment with another process running on the same processor. For example, process $X can share a segment with any of these processes on the same processor:

  - Any process that has the same process access ID (PAID)
  - Any process that has the same group ID, if $X is the group manager (that is, if $X has a PAID of *group-id*,255)
  - Any process, if $X has a PAID of the super ID (255,255)

  If processes are running in different processors, they can share a segment only if the security requirements are met and the segment is a read-only segment.

  Callers of ALLOCATESEGMENT can share segments with callers of SEGMENT_ALLOCATE_. High-PIN callers can share segments with low-PIN callers.

- Sharing flat segments

  A process cannot share a flat segment with a process that allocated a selectable segment, because the segments reside in different parts of memory. (Similarly, a process cannot share a selectable segment with a process that allocated a flat segment.)

  For shared flat segments, the call to ALLOCATESEGMENT can be followed by a call to USESEGMENT, but calling USESEGMENT is unnecessary because all of the flat segments allocated by a process are always accessible to the process.

  For more information on flat segments, see the SEGMENT_ALLOCATE_ Procedure on page 14-5.

Examples

```
STATUS := ALLOCATESEGMENT (SEGMENT^ID, SEG^SIZE, SWAP^FILE);
          !  standard call to create a user segment;
          !  "swap^file" parameter can be omitted

STATUS := ALLOCATESEGMENT (SEGMENT^ID, , FILENAME, %60000);
          ! allocates a read-only segment whose
          ! segment size is taken from the size of the
          ! swap file
```

```
STATUS := ALLOCATESEGMENT (SEGMENT^ID, SEGMENT^SIZE,
                           FILENAME, %44000);
          ! allocates an extensible segment whose swap file
          ! disk extents will be allocated as needed

STATUS := ALLOCATESEGMENT (SEGMENT^ID, , , PIN);
          ! allocates a shared segment, which is shared
          ! using the PIN method with the segment given by
          ! SEGMENT^ID in the process identified by PIN

STATUS := ALLOCATESEGMENT (SEGMENT^ID, , FILENAME, %50000);
          ! allocates a shared segment, shared using the
          ! file name method
```

ALTER Procedure (Superseded by FILE_ALTERLIST_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The ALTER procedure changes certain attributes of a disk file that are normally set when the file is created.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
CALL ALTER ( file-name                ! i
           ,function                  ! i
           ,newvalue                  ! i
           ,[ partonly ] );           ! i
```

Parameters

file-name input

INT:ref:12

is an array containing the internal-format file name of the disk file to be altered.

function input

INT:value

is a value specifying what characteristic of the file is to be changed. See Table 2-2 on page 2-29.

newvalue input

INT:ref:*

is an integer array supplying the new value for the characteristic specified by *function*. Its size is dependent on the operation. See Table 2-2 on page 2-29.

partonly input

INT:value

if present, specifies for partitioned files whether the function is to be performed for all partitions of the file (if the value given is zero), or just for the named partition (if the value is one). Nonpartitioned files should use zero.

If omitted, zero is assumed. A value of one cannot be specified for some *function*s, as noted below. If a function would affect alternate key files, then a value of one will prevent this.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILEINFO or FILE_GETINFO_).

= (CCE) indicates that the call to ALTER was successful.

> (CCG) indicates that an error occurred (call FILEINFO or FILE_GETINFO_).

Considerations

- The file cannot be opened when ALTER is called.
- The security on the file must allow the caller to have read and write access.
- If the characteristic already has the supplied value, no error is indicated.
- The ALTER procedure supports format 2 files, except for changing alternate key or partition descriptions (functions 5 and 6 of Table 2-2).
- Except as noted in the table below, the alterations are not made to alternate key files, but are made to secondary partitions of a partitioned file unless the value of *partonly* is 1.
- A secondary partition can be changed only if the value of *partonly* is 1.
- If a partition (or alternate key file) is not accessible, error 3 (or 4) will result, but the accessible partitions (or files) will still be updated.
- If the secondary partition of the file is audited differently from the primary (one is audited and the other is not), error 80 is returned and you cannot alter the audit flag.

Table 2-2. ALTER Function Codes (page 1 of 2)

Code	Description
1	File-code: Change the application defined file code associated with the file. File codes 100-999 are reserved for use by HP. The *newvalue* parameter should be a one-word binary number.
2	Audited: Change the TMF audited characteristic of the file (*file-type*.<2> from FILEINFO or item 66 from FILE_GETINFOLIST[BYNAME]_). The *newvalue* parameter should be a 1-word binary number with a value of 1 to make the file audited or a value of 0 to make it unaudited. Unless the value of *partonly* is 1, all alternate key files as well as all partitions will be changed.
3	Refresh: Change the flag controlling whether the file's EOF value is written out each time it is changed (*file-type*.<10> from FILEINFO or item 70 from FILE_GETINFOLIST[BYNAME]_). The *newvalue* parameter should be a 1-word binary number with a value of 1 to cause writing or 0 to avoid writing.

Table 2-2. ALTER Function Codes (page 2 of 2)

Code	Description
4	Oddunstr: For an unstructured file, make the file allow odd byte positioning and transfers (indicated by *file-type*.<12> from FILEINFO or item 65 from FILE_GETINFOLIST[BYNAME]_). The *newvalue* parameter must be a one-word binary number with value of 1. Once set for a file, this characteristic cannot be reset.
5	Alternate Keys: For a structured file, change the alternate key description. The *newvalue* parameter should be an array in the same format as the *alternate-key-params* array of the CREATE procedure. This function changes only the description in the primary file; no alternate key files are purged or created. The *partonly* parameter must be zero for this function. This function is not supported for format 2 files.
6	Partitions: Change the partitioning description of the file. The *newvalue* parameter should be an array in the same format as the *partition-params* array of the CREATE procedure. The partition description can be changed only in these ways: a) The volume name of an existing partition may change. b) For a key-sequenced file, the extent sizes of a partition may be changed. c) For non-key-sequenced files, new partitions may be added. This function changes the description only in the primary file; no secondary partitions are moved, updated, or created. The *partonly* parameter must be zero for this function. This function is not supported for format 2 files.
7	Broken Flag: Resets the broken flag (which is shown in *open-flags2*.<6> of FILEINFO or item 78 from FILE_GETINFOLIST[BYNAME]_). The *newvalue* parameter must be a one-word binary number with a value of 0. For a partitioned file, the *partonly* parameter must have a value of 1 for this function.
8	Expiration Date: Change the expiration date associated with the file to the one given in *newvalue*. The *newvalue* parameter must be a 4-word GMT timestamp. The expiration date is not changed for associated alternate key files, but is changed for the secondary partitions of a partitioned file (unless the value of *partonly* is 1). The expiration date for a temporary file cannot be set with ALTER, since temporary files must be purged when closed.

OSS Considerations

- This procedure operates only on Guardian objects. If an OSS file is specified, error 1163 occurs.

Example

```
INT fname[0:11] := ["$data   foo     bar     "];
INT change^filecode := 1;
```

```
INT newcode := 101;
      .
      .
CALL ALTER( fname, change^filecode, newcode );! see Table 2-2
IF <> THEN ...
```

ALTERPRIORITY Procedure (Superseded by PROCESS_SETINFO_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The ALTERPRIORITY procedure is used to change the execution priority of a process or process pair.

A process or process pair has two priority values: the initial priority value and the current priority value. ALTERPRIORITY changes both priority values to the specified value.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
CALL ALTERPRIORITY ( process-id                    ! i
                   ,priority );                    ! i
```

Parameters

process-id input

INT:ref:4

is a 4-word array containing the process ID of the process whose execution priority is to be changed, where:

`[0:2]`	Process name or creation timestamp
`[3].<0:3>`	Reserved
`.<4:7>`	Processor number where the process is executing
`.<8:15>`	PIN assigned by the operating system to identify the process in the processor

If *`process-id`*[0:2] references a process pair and *`process-id`*[3] is specified as -1, then the call applies to both members of the process pair.

`priority` input

INT:value

is a new execution priority value in the range of {1:199} for *`process-id`*.

Condition Code Settings

< (CCL)	indicates that ALTERPRIORITY failed, or no process designated as *`process-id`* exists.
= (CCE)	indicates that the priority of the process is altered.
> (CCG)	does not return from ALTERPRIORITY.

Considerations

When ALTERPRIORITY is called on a Guardian process, the caller must be either the super ID (255,255), the group manager (n,255) of the process access ID, or a process with the same process access ID as the process or process pair whose priority is being changed. For the PROCESSACCESSID procedure, see “Considerations” and for further information about the process access ID, see the *Guardian User’s Guide*.

When ALTERPRIORTY is called on an OSS process, the security rules that apply are the same as those that apply to calling the OSS `kill()` function. See the `kill(2)` function reference pages either online or in the *Open System Services System Calls Reference Manual* for details.

ARMTRAP Procedure (Superseded by SIGACTION_INIT_Procedure)

Summary

Note. This procedure cannot be called by OSS or native processes; use the signal procedures. TNS Guardian processes must continue to use this procedure.

The ARMTRAP procedure is used to specify a trap handler (that is, a location within the application program where execution begins if a trap occurs) and also to return from a trap handler.

Syntax for C Programmers

```
#include <cextdecs(ARMTRAP)>

void ARMTRAP ( short traphandlr-addr
              ,short trapstack-addr );
```

There are restrictions on calling the ARMTRAP procedure from a C program due to the fact that all C programs run under the Common Run-Time Environment (CRE).

It is not possible to express a trap handler in C. The use of ARMTRAP in C programs is limited to calling ARMTRAP(-1,-1) to turn off trap handling by overriding the trap handler installed by the CRE in a TNS Guardian C program.

For details, see the *Common Run-Time Environment (CRE) Programmer's Guide*.

Syntax for TAL Programmers

```
CALL ARMTRAP ( traphandlr-addr                    ! i
              ,trapstack-addr );                  ! i
```

Parameters

traphandlr-addr input

INT:value

is a label (nonzero P register value) that identifies a statement in the program where control is to transfer if a trap occurs.

You can specify 0 for *traphandlr-addr* only in a call from within a trap handler. Such a call causes the process to resume. For details, see Considerations on page 2-36.

trapstack-addr input

INT:value

is an address specifying the local data area for the application process's trap handler. The *trapstack-addr* parameter also indicates where the trap number and stack marker at the time of the trap are passed to the application process.

If *trapstack-addr* has a value < 0, then trap handling is disabled and any trap results in the process being stopped with a process deletion (ABEND) message.

ARMTRAP Functions

Use the ARMTRAP procedure to perform one of these functions:

- Arm a trap handler (that is, specify a location in the application program where execution begins if a trap occurs). To do this, set *traphandlr-addr* to a value greater than 1 specifying the address of a label at which the trap handler starts and set *trapstack-addr* to a nonnegative value specifying the stack address above which its activation record will go.
- Set default handling (that is terminate, but enter the debugger if in a debug session). To do this, set *traphandlr-addr* to one and set *trapstack-addr* to zero.
- Set traps to enter the debugger. To do this, set *traphandlr-addr* to one and *trapstack-addr* to a value greater than zero. By convention, both parameters are normally set to 1 in such as call.
- Disarm all trap handling (that is, specify that no part of the application program is to execute if a trap occurs). To do this, set *traphandlr-addr* to a nonzero value and set *trapstack-addr* to a negative value. By convention, both parameters are normally set to -1 in such a call.
- Resume the process and rearm the trap handler. This must be done by a call to ARMTRAP from within a trap handler with *traphandlr-addr* set to 0 and *trapstack-addr* set to a nonnegative value specifying the stack address above which its activation record will go. For details, see Considerations on page 2-36.
- Resume the process and disarm all trap handling. This must be done by a call to ARMTRAP from within a trap handler with *traphandlr-addr* set to 0 and *trapstack-addr* set to a negative value. For details, see Considerations on page 2-36.

Trap Handler Activation and Termination

When a trap handler has been armed and a trap subsequently occurs, control passes to the trap handler at the location specified by *traphandlr-addr* in the same code segment as the original call to ARMTRAP. Trap handling is automatically disabled. 'S' and 'L' are set to *trapstack-addr* plus 6; the seven words starting at *trapstack-addr* are (relative to the new 'L' setting):

'L'[-6] Reserved

'L'[-5] Space ID at the time of the trap

'L'[-4] Trap number:

0 Invalid address reference
1 Instruction failure
2 Arithmetic overflow
3 Stack overflow
4 Process loop timer timeout
5 Call from process with PIN > 255
11 Memory manager read error
12 No memory available
13 Uncorrectable memory error

'L'[-3] Value of ‘S’ at the time of the trap; it is -1 if the trap occurs in a protected code area (see “Considerations”)

'L'[-2] Value of ‘P’ at the time of the trap; the ‘P’ value associated with the space ID in ‘L’[-5] completely identifies the location of the trap

'L'[-1] Value of the hardware ENV register at the time of the trap

'L'[0] Value of ‘L’ at the time of the trap

The locations ‘L’[-5] through ‘L’[0] are referred to as trap variables: space ID, trap number, S, P, ENV, and L, respectively.

The trap handler exits by a call to ARMTRAP with *trapstack-addr* = 0. The process’s registers at the time that it resumes are set to the values indicated by these ‘L’ relative locations:

'L'[-6] Reserved

'L'[-5] New value for space index, in bits <11:15>; bits <0:10> are ignored (see “Considerations”)

'L'[-4] Ignored

'L'[-3] New value for S register

'L'[-2] New value for P register

'L'[-1] New value for hardware ENV register

'L'[0] New value for L register

'L'[1] New value for R0

'L'[2] New value for R1

'L'[3] New value for R2

'L'[4] New value for R3

'L'[5] New value for R4

'L'[6] New value for R5

'L'[7] New value for R6

'L'[8] New value for R7

Note that parts of 'L'[-5] and 'L'[-1] are combined into the new space ID.

Considerations

- Space required for a trap handler

  Typically the *trapstack-addr* value activates the trap handler near the high end of the process stack. At least 350 words must be available between the trap address value specified to ARMTRAP and either the last word in the application's data area or 'G'[32767], whichever is less. Alternatively, stack space for the trap handler can be allocated among the process global variables, below the stack.

- Saving the register stack and allocating local data

  Upon entry to the application process's trap handler, the stack registers (R0-R7) contain the values they had at the time of the trap. To save these values, the first statement of the trap handler must be:

  ```
  CODE ( PUSH %777 )
  ```

  This saves the register stack contents. Local storage can then be allocated by adding the appropriate value to 'S' through a statement of the form:

  ```
  CODE ( ADDS num-locals )
  ```

  where *num-locals* is a LITERAL defining the number of words of local storage needed (see the next consideration).

- Base-address equivalencing and declaring local variables

  Any local variables in the application program's trap-handling procedure must be declared relative to the L register by using base-address equivalencing. For example:

  ```
  INT I = 'L' + 9;
  STRING .EXT X(X_TEMPLATE) = 'L' + 12;
  ```

  Assuming that the trap handler begins with the statement CODE (PUSH %777), the first local variable should be placed at 'L'+9.

  For details on base-address equivalencing, see the *TAL Reference Manual*.

  Note. Variables declared in this form cannot be initialized.

  The trap-handling procedure must contain a statement that explicitly allocates storage for any locally declared variables (see the preceding consideration).

- Space ID, P register, and hardware ENV register

  The space ID consists of a code space bit that signifies system or user space, a library space bit that signifies code or library space, and a 5-bit index to indicate

which of the 32 possible segments you are referring to. For more information about space IDs, see the appropriate system description manual for your system.

At the time of a trap, the space ID of the calling procedure is placed in the stack at 'L'[-5]; the stack marker ENV register at 'L'[-1] also contains the system code and library code bits but not the index. When exiting the trap handler, execution resumes at the location identified by the P register at 'L'[-2], the space index in the stack at 'L'[-5], and the library space and code space bits of 'L'[-1].

- Value for the P register

  The value for the P register at the time of the trap depends upon the trap condition. In this table, I represents the address of the instruction being executed at the time of the trap, and ? means undefined.

Trap	P Register
0	I
1	I
2	I + 1
3	?
4	I
5	?
11	I
12	I
13	?

- Terminating a trap handler

  A trap handler can terminate in these ways on either a TNS or native system:

  - Clear the overflow (or trap) bit in the trap ENV variable and resume from a trap 2 (arithmetic overflow). See Resuming at the point of the trap on page 2-38. (If the overflow and trap bits of ENV are both set upon exit from the trap handler, then on TNS systems, another overflow trap immediately occurs and on native systems, the process abends.)
  - Resume with no modifications (modifying the overflow or trap bit of the trap ENV variable is permitted) after a trap 4 (loop timer interrupt). See Resuming at the point of the trap on page 2-38.
  - Jump to a restart point by changing the trap variables P, L, ENV, space ID, and S. See Resuming at the point of the trap on page 2-38.
  - Terminate the process (for example, by a call to PROCESS_STOP_).

  Attempting to exit from a trap handler in any other way is not recommended; the results are likely to vary between TNS and native systems.

- Resuming from (exiting) a trap handler

The only way to exit from a trap handler is by a call to ARMTRAP specifying *traphandlr-addr* = 0; the value supplied for *trapstack-addr* determines whether the trap handler is rearmed or disarmed when program execution resumes. The trap handler must use ARMTRAP. (It cannot use an EXIT instruction to exit through the stack marker at the current L register location; using EXIT would result in an invalid S register setting following the exit and would leave trap handling disabled.)

If a call to ARMTRAP is made from within a trap handler and if a value other than 0 is specified for *traphandlr-addr*, the trap handler continues to execute. The result of such a call is:

- If the call specifies a new trap handler (by supplying a nonzero value for *trapstack-addr*), the new trap handler is not armed until a call with *traphandlr-addr* = 0 is made that explicitly arms it.
- If the call disables trap handling (by specifying *trapstack-addr* < 0), traps remain disabled even after a call with *traphandlr-addr* = 0 that would normally rearm them.

A call to ARMTRAP with *traphandlr-addr* = 0 is invalid if not made from within a trap handler. Starting in D20, such a call disarms all trap handling.

- Resuming at the point of the trap

  To resume execution at the point of the trap, the trap handler should not modify any of the values passed to it except, under some circumstances, the overflow bit or trap bit of the trap ENV variable at 'L'[-1]. Such resumption is valid only for trap 2 (arithmetic overflow) or trap 4 (loop timeout). An attempt to resume at the point of any other trap typically causes the same trap to occur again on a TNS system; on a native system, such an attempt causes the process to abend.

- Resuming at another point in the program

  To resume execution at some other point in the program, you need to change the P register value at 'L'[-2], the space index at 'L'[-5], and, if necessary, the library space bit in ENV at 'L'[-1] to reflect the new location within your program. You also need to set appropriate values for the S register at 'L'[-3] and the L register at 'L'[0] and the appropriate environment state in ENV at 'L'[-1]: The RP field (ENV.<13:15>) should be set to 7 if the resumption point is the beginning of a statement; ENV.<0> should be set to 0.

- Traps in protected code

  If the trap occurs in system code or system library and the trap handler is in user code or user library, or if the trap occurs in a licensed user library and the trap handler is in user code, the reported program location and process state (space ID, P, ENV, and L) indicate the point of the user call to one of these protected code regions and S is reported as -1.

- How to avoid writing over the application's data stack

If ‘L’[-3] (the value of ‘S’ at the time of the trap) is -1, the trap handler should not resume from the trap handler without first changing ‘L’[-3] to a more appropriate value. Otherwise, ‘G’[0] through ‘G’[10] of the application’s data stack are overwritten.

- When the trap handler is not invoked

  Under some circumstances (for example, if system resources that are necessary to initiate trap handling are not available), the trap handler specified though ARMTRAP might not execute. In such a case, the process abends.

Additional Considerations for Native Systems

Special restrictions apply to trap handlers that execute on native systems. These rules should be observed:

- Trap P variable

  The TNS trap P variable is only approximate for a process running in accelerated mode. You should not use it to inspect the code area and determine the failing instruction.

  You should not increment the trap P variable and resume execution; doing so causes undefined results. However, you can change the trap P variable to a valid TNS restart point. The restart point would typically be a label in your program. See Resuming at another point in the program on page 2-38

- Invalid trap ENV fields

  For a process running in accelerated mode, the ENV field RP is not valid and the fields N, Z, and K are not reliable.

- Register stack R[0:7]

  The contents of the TNS register stack are not valid in accelerated mode and are not dependable in TNS mode. You should never change the register stack when attempting to resume at the point of the trap.

- Functions

  A trap-handling procedure must not be a function returning a result value.

OSS Considerations

Do not use the ARMTRAP procedure in OSS processes.

△ **Caution.** Use of this procedure in an OSS process causes undefined results and might cause severe side effects such as disabling signals completely or causing the calling process to receive a fatal signal.

Example

```
PROC TRAPPROC;
  BEGIN
  CALL ARMTRAP ( @TRAP, $LMIN ( LASTADDR , %77777 ) - 500 );
    ! setting the trap.
  RETURN;
TRAP:
    CODE (PUSH %777);
    .
    .
    .
    CALL ARMTRAP ( 0, $LMIN ( LASTADDR , %77777 ) - 500 );
  END;

PROC MAIN PROC;
  BEGIN
    CALL TRAPPROC;
       .
       .
       .
  END;
```

In the previous example, @TRAP is the label at the beginning of the Transaction Application Language (TAL) trap-handling procedure where control is transferred if a trap occurs. The $LMIN expression is the address of the local data area where the trap handler runs (its data area). The second call to ARMTRAP is the return from the trap handler.

Related Programming Manual

For programming information about the ARMTRAP trap-handling procedure, see the *Guardian Programmer's Guide*.

AWAITIO[X|XL] Procedures

Summary

The AWAITIO, AWAITIOX, and AWAITIOXL procedures are used to complete a previously initiated I/O operation. Use AWAITIOX[L] with the extended 32-bit (or 'X') versions of the I/O procedures such as READX, WRITEREADX, and so forth. Use

AWAITIO with the 16-bit versions such as READ, WRITEREAD, and so forth. Use AWAITIO[X|XL] to:

- Wait for the operation to complete on:
  - A particular file—Application process execution suspends until the completion occurs. A timeout is considered to be a completion in this case.
  - Any file or for a timeout to occur—A timeout is not considered a completion in this case.
- Check for the operation to complete on:
  - A particular file—The call to AWAITIO[X|XL] immediately returns to the application process, regardless of whether there is a completion or not. (If there is no completion, an error indication is returned.)
  - Any file

If AWAITIO[X|XL] is used to wait for a completion, you can specify a time limit.

Syntax for C Programmers

```
#include <cextdecs(AWAITIO)>

_cc_status AWAITIO ( short _near *filenum
              ,[ short _near *buffer-addr ]
              ,[ unsigned short _near *count-transferred ]
              ,[ __int32_t _near *tag ]
              ,[ __int32_t timelimit ] );


#include <cextdecs(AWAITIOX)>

_cc_status AWAITIOX ( short _far *filenum
                  ,[ __int32_t _far *buffer-addr ]
                  ,[ unsigned short _far *count-transferred ]
                  ,[ __int32_t _far *tag ]
                  ,[ __int32_t timelimit ]
                  ,[ short _far *segment-id ] );


#include <cextdecs(AWAITIOXL)>


short  AWAITIOXL ( short _far *filenum
                  ,[ __int32_t _far *buffer-addr ]
                  ,[ __int32_t _far *count-transferred ]
                  ,[ long long _far *tag ]
                  ,[ __int32_t timelimit ]
                  ,[ short _far *segment-id ] );
```

- The function value returned by AWAITIO[X], which indicates the condition code, can be interpreted by the `_status_lt()`, `_status_eq()`, or `_status_gt()` macros (defined in the file `tal.h`)

Syntax for TAL Programmers

```
CALL AWAITIO[X] ( filenum                  ! i,o
                 ,[ buffer-addr ]          ! o
                 ,[ count-transferred ]    ! o
                 ,[ tag ]                  ! o
                 ,[ timelimit ]            ! i
                 ,[ segment-id ] );        ! o   (AWAITIOX only)


error:= AWAITIOXL ( filenum                ! i,o
                 ,[ buffer-addr ]          ! o
                 ,[ count-transferred ]    ! o
                 ,[ tag ]                  ! o
                 ,[ timelimit ]            ! i
                 ,[ segment-id ] );        ! o
```

Parameters

error returned value

INT (Use with AWAITIOXL)

is a file-system error number indicating the outcome of the operation.

0 (FEOK)

indicates a successful operation.

filenum input, output

INT:ref:1 (Use with AWAITIO)
INT .EXT:ref:1 (Use with AWAITIOX[L])

is the number of an open file. If a particular *filenum* is passed, AWAITIO[X|XL] applies to that file.

If *filenum* is passed as -1, the call to AWAITIO[X|XL] applies to the oldest incomplete operation pending on each file. The specific action depends on the value of the *timelimit* parameter (see the *timelimit* parameter below).

AWAITIO[X|XL] returns into *filenum* the file number associated with the completed operation.

buffer-addr output

WADDR:ref:1 (Use with AWAITIO)

returns the address of the *buffer* specified when the operation was initiated.

EXTADDR .EXT:ref:1 (Use with AWAITIOX and AWAITIOXL)

returns the relative extended address of the *buffer* specified when the operation was initiated.

If the actual parameter is used as an address pointer to the returned data and is declared in the form INT .EXT *buffer-addr*, it should be passed to AWAITIO[X|XL] in the form @*buffer-addr*.

count-transferred output

INT:ref:1 (Use with AWAITIO)
INT .EXT:ref:1 (Use with AWAITIOX)
INT(32).EXT:ref:1 (Use with AWAITIOXL)

returns the count of the number of bytes transferred because of the associated operation.

tag output

INT(32):ref:1 (Use with AWAITIO)
INT(32) .EXT:ref:1 (Use with AWAITIOX)
INT(64) .EXT:ref:1 (Use with AWAITIOXL)

returns the application-defined tag that was stored by the system when the I/O operation associated with this completion was initiated. The value of *tag* is undefined if no tag was supplied in the original I/O call. If the completed I/O operation has a 32-bit tag, the 64-bit tag is in the sign-extended value of the 32-bit *tag*.

timelimit input

INT(32):value (Use with AWAITIO, AWAITIOX, and AWAITIOXL)

indicates whether the process waits for completion instead of checking for completion. If *timelimit* is passed as:

>	0D	A wait-for-completion is specified. The *timelimit* parameter specifies the maximum time (in .01-second units) from the time of the AWAITIO[X|XL] call that the application process can wait (that is, be on a wait list) for completion of a waited-for operation. See “Considerations” for queue files.
=	-1D	An indefinite wait is indicated.
=	0D	A check for completion is specified. AWAITIO[X|XL] immediately returns to the caller, regardless of whether or not an I/O completion occurs.
<	-1D	File-system error 590 occurs.
omitted		An indefinite wait is indicated.

segment-id output

INT .EXT:ref:1 (Use with AWAITIOX and AWAITIOXL)

returns the segment ID of the extended data segment containing the *buffer* when the operation was initiated. If the *buffer* is not in a selectable segment, *segment-id* is -1.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILE_GETINFO_ or FILEINFO).

= (CCE) indicates that an I/O operation completed.

> (CCG) indicates that an I/O operation completed but a warning occurred (call FILE_GETINFO_ or FILEINFO).

Considerations

- Completing nowait calls

  Each nowait operation initiated must be completed with a corresponding call to AWAITIO[X|XL].

  - If AWAITIO[X|XL] is used to wait for completion (*timelimit* <> 0D) and a particular file is specified (*filenum* <> -1), completing AWAITIO[X|XL] for any reason, except interruption by an OSS signal, is considered a completion: if the I/O operation did not complete, error 40 is returned and the oldest I/O operation against the file is canceled.

  - Queue files

    If a nowait READUPDATELOCK[X] operation is used in conjuction with the AWAITIO[X|XL] *timelimit* > 0D, this occurs:

    If the queue file timeout occurs before the time limit, the read request is completed with error 162.

    If the time limit expires before the queue file timeout, the READUPDATELOCK[X] request is canceled. A canceled READUPDATELOCK[X] can result in the loss of a record from the queue file. If the time limit expires before the queue file timeout, the READUPDATELOCK[X] request is canceled if it was a file-specific call (that is, the file number is other than -1). With non file-specific calls, READUPDATELOCK[X] is not canceled for the queue file. A canceled READUPDATELOCK[X] can result in the loss of a record from the queue file. For audited queue files, record loss can be avoided by performing an ABORTTRANSACTION procedure, when detecting error 40, to ensure that any

dequeued record is reinserted into the file. For nonaudited queue files, there is no means of assuring recovery of a lost record. Thus, your application should never call AWAITIO[X|XL] with a time limit greater than 0D if READUPDATELOCK[X] is pending. The ABORTTRANSACTION recovery procedure does not work on nonaudited queue files.

- If AWAITIO[X|XL] is used to check for completion (*timelimit* = 0D) or used to wait on any file (*filenum* = - 1), completing AWAITIO[X|XL] does not necessarily indicate a completion.

If you perform an operation using one of these procedure calls with a file opened nowait, you must complete the operation with a call to the AWAITIO[X|XL] procedure:

```
CONTROL                     SETMODENOWAIT
CONTROLBUF
                            UNLOCKFILE
LOCKFILE                    UNLOCKREC
LOCKREC
                            WRITE[X]
READ[X]                     WRITEREAD[X]
READLOCK[X]                 WRITEUPDATE[X]
READUPDATE[X|XL]            WRITEUPDATEUNLOCK[X]
READUPDATELOCK[X]
```

Note. Use AWAITIO only with the 16-bit I/O versions of the above procedures, such as READ, WRITEREAD, and so forth. You can use AWAITIOX with any versions of the above procedures, including READX, WRITEREADX, and so forth. You can use AWAITIOXL with SERVERCLASS_SENDL_ , READUPDATE[X|XL], and any versions of the above procedures.

- Completion *tag* values

  A *tag* -30D returned by AWAITIO signals completion of a nowait open; a *tag* -29D returned by AWAITIO signals completion of a nowait backup open. For more information, see the FILE_OPEN_CHKPT_ Procedure on page 5-130.

- Using AWAITIO, AWAITIOX, and AWAITIOXL

  Nowait calls to the extended I/O routines must call AWAITIOX or AWAITIOXL to complete the operation. AWAITIOX and AWAITIOXL also completes calls made to the 16-bit I/O routines. Thus, you can replace all current calls to AWAITIO and the calls to AWAITIOX with call to AWAITIOXL.

  If the operation was initiated with a call to READ, WRITEREAD, and so on (the 16-bit I/O routines), and AWAITIOX or AWAITIOXL is called to complete the operation, *buffer-addr* contains the extended address of that *buffer* and *segment-id* is -1.

  If you accidentally call AWAITIO and extended I/O operations are outstanding against the file, AWAITIO does not complete the operation. If you call AWAITOX while an "L" operation (for example, SERVERCLASS_SENDL_) is outstanding, AWAITIOX does not complete the operation. If a specific file number is given, error 2 is returned. You must then call AWAITIOX or AWAITIOXL to complete the

operation. If the file number was -1, the files with extended I/O operations outstanding are skipped and AWAITIO will check the completion of any 16-bit I/O operations still outstanding.

- Reference parameters for AWAITIOX and AWAITIOXL

  The reference parameters for AWAITIOX and AWAITIOXL can be in the user's stack or in an extended data segment. The reference parameters cannot be in the user's code space.

  The reference parameters for AWAITIOX and AWAITIOXL must be relative extended addresses; they cannot be absolute extended addresses.

  If the reference parameters for AWAITIOX and AWAITIOXL address an area in a selectable extended data segment, the segment must be in use at the time of the call to AWAITIOX and AWAITIOXL. (Flat segments allocated by a process are always accessible to the process.)

- AWAITIOX or AWAITIOXL and *buffer* in extended data segment

  If the *buffer* is in a flat extended data segment, the segment must be allocated at the time of the call to AWAITIOX or AWAITIOXL.

  If the *buffer* is in a selectable extended data segment, the segment need not be in use at the time of the call to AWAITIOX or AWAITIOXL. However, the segment must be allocated at the time of the call to AWAITIOX or AWAITIOXL.

- Normal order of I/O completion (without SETMODE 30)

  If SETMODE 30 is not set, the oldest incomplete I/O operation always completes first; therefore, AWAITIO[X|XL] completes I/O operations associated with the particular open of a file in the same order as initiated.

- Order of I/O completion with SETMODE 30

  Specifying SETMODE 30 allows nowait I/O operations to complete in any order. However, I/O operations that complete at the same time return in the order issued (unless SETMODE 30 is specified with *param1* set to 3). An application process that uses this option can use the *tag* parameter to keep track of multiple I/O operations associated with a file open.

- Operation timed out

  If an error indication is returned on a call where either *timelimit* = 0 or *filenum* = -1 was specified, and a subsequent call to FILE_GETINFO_ or FILEINFO shows that an error 40 occurred, the operation is considered incomplete and AWAITIO[X|XL] must be called again.

- Write buffers

The contents of a buffer should not be altered between the initiation of a nowait I/O operation (for example, a call to WRITE[X]) and the completion of that operation (that is, a call to AWAITIO[X|XL]).

▲ **WARNING.** Modifying nowait WRITE buffers before the AWAITIOX that completes WRITE can cause data corruption to or from the opened file.

However, you can alter the contents of a buffer if set SETMODE 72,1 is called. For more information, see Setmode 72 on page 14-80.

- Read buffers

  If the file was opened by FILE_OPEN_ , or if it was opened by OPEN and SETMODE 72 was called with *param1* set to 0, the buffer used for a read operation should not be used for any other purpose (including another read) until the read operation has been completed with a call to AWAITIO[X|XL].

▲ **WARNING.** Modifying nowait READ buffers before the AWAITIOX that completes READ can cause data corruption to or from the opened file.

- No nowaited operations

  You should not call AWAITIO[X|XL] unless you initiate a nowait operation before the call. AWAITIOXL returns the error 26 code directly and does not return CCL. Otherwise, a subsequent call to FILE_GETINFO_ or FILEINFO shows that an error 26 occurred.

- Error handling

  AWAITIOXL returns the error code directly. For AWAITIO[X], pass the file number returned by AWAITIO[X] to the FILE_GETINFO_ or FILEINFO procedure to determine the cause of the error. If *filenum* = -1 (that is, any file) is passed to AWAITIO[X|XL] and an error occurs on a particular file, AWAITIO[X|XL] returns the file number associated with the error in *filenum*.

- AWAITIO[X|XL] and edit files

  If AWAITIO[X|XL] returns after completion of an I/O operation against an EDIT file that was accessed using the IOEdit procedures, you must call the COMPLETEIOEDIT procedure to inform the IOEdit software that the operation has finished.

- AWAITIO[X|XL] completion summary

  How AWAITIO[X|XL] completes depends on whether the *filenum* parameter specifies a particular file or any file and on what the value of *timelimit* is when passed with the call. The action taken by AWAITIO[X|XL] for each combination of *filenum* and *timelimit* is summarized in Table 2-3.

- AWAITIO[X|XL] operation

  The operation of the AWAITIO[X|XL] procedure is shown in Figure 2-1.

Table 2-3. AWAITIO[X|XL] Action

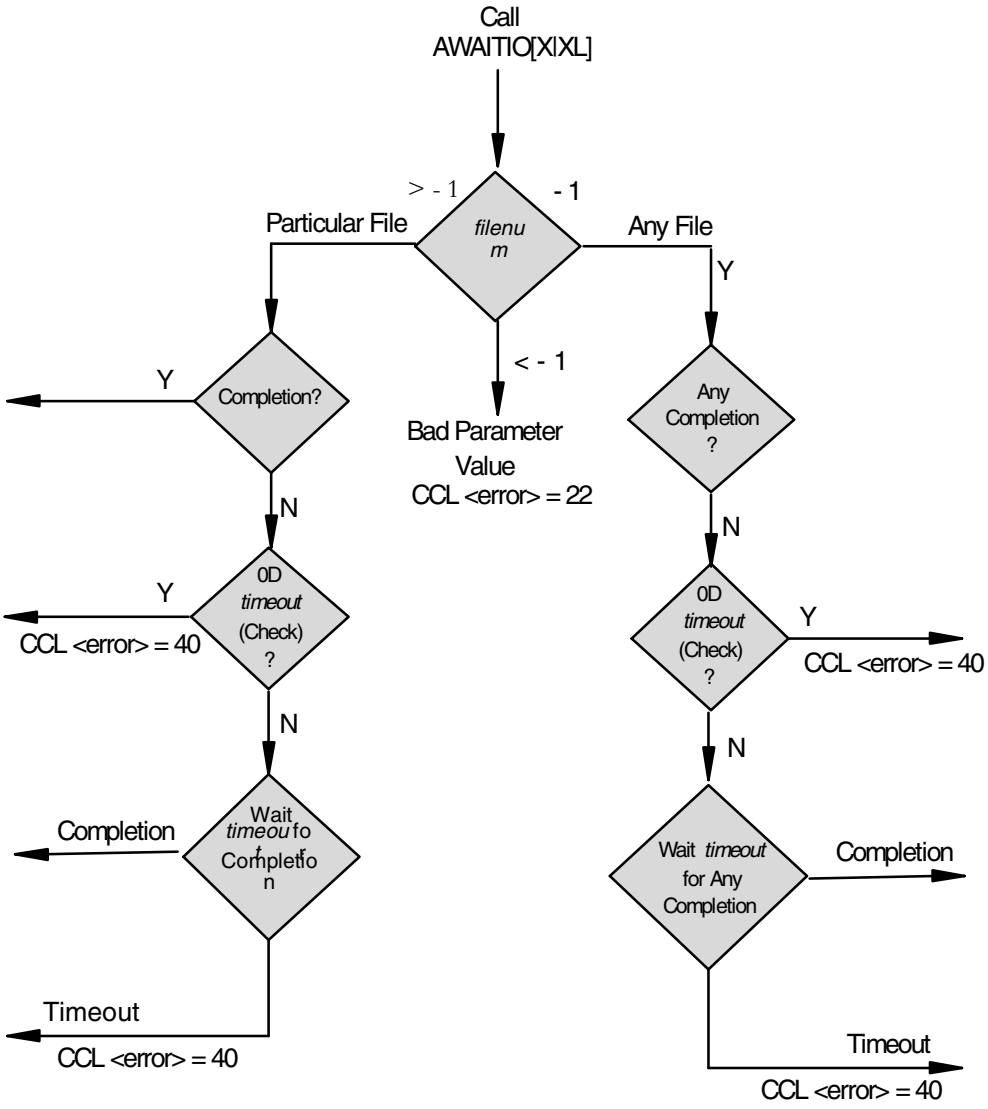
	***timelimit* = 0**	***timelimit* <> 0**
Particular File *filenum* = a file number	CHECK for any I/O completion on *filenum*. COMPLETION File number is returned in *filenum*. Tag of completed call is returned in *tag*. NO COMPLETION CCL (error 40) is returned. File number returned is in *filenum*. No I/O operation is canceled.	WAIT for any I/O completion on *filenum*. COMPLETION File number is returned in *filenum*. Tag of completed call is returned in *tag*. NO COMPLETION CCL (error 40) is returned. File number is returned in *filenum*. Oldest I/O operation on *filenum* is canceled. Tag of canceled call is returned in *tag*.
Any File *filenum* = –1	CHECK for any I/O completion on any open file. COMPLETION File number of completed call is returned in *filenum*. Tag of completed call is returned in *tag*. NO COMPLETION CCL (error 40) is returned. The value –1 is returned in *filenum*. No I/O operation is canceled.	WAIT for any I/O completion on any open file. COMPLETION File number of completed call is returned in *filenum*. Tag of completed call is returned in *tag*. NO COMPLETION CCL (error 40) is returned. The value –1 is returned in *filenum*. No I/O operation is canceled.

Note:

This table assumes that SETMODE 30 has been set.

AWAITIOXL returns the error code directly.

Figure 2-1. AWAITIO[X|XL] Operation



Note. AWAITIOXL returns the error code directly.

OSS Considerations

When an OSS process calls AWAITIO[X|XL] and an OSS signal occurs, the OSS function completes with error 4004 (EINTR). Even if AWAITIO[X|XL] is used to wait for completion (*timelimit* <> 0D) and a particular file is specified (*filenum* <> -1), this is not considered a completion and the oldest I/O operation against the file is not canceled. You should call AWAITIO[X|XL] again to complete the I/O operation.

Related Programming Manual

For programming information about the AWAITIO[X] file-system procedures, see the *Guardian Programmer's Guide*.

Note. The AWAITIOXL procedure is supported on systems running J06.07 and later J-series RVUs and H06.18 and later H-series RVUs.

BACKSPACEEDIT Procedure

Summary

The BACKSPACEEDIT procedure sets the current record number of an IOEdit file to that of the line preceding what was the current record before the call. These rules describe the operation of BACKSPACEEDIT:

- If the current record number is -1 (which occurs when the file is empty or when the file is positioned at the beginning, as when it is just opened), BACKSPACEEDIT does nothing.
- If the current record number is -2 (which occurs when the current record number has been incremented beyond the last record in the file), the current record number is set to the highest record number in the file. If the file is empty, the current record number is set to -1.
- If the current record number is 0 or greater, the current record number is set to the highest record number in the file that is less than the current record number before the call; if there is no such record, the current record number is set to -1.

BACKSPACEEDIT is an IOEdit procedure and can only be used with files that have been opened by OPENEDIT or OPENEDIT_.

Syntax for C Programmers

```
#include <cextdecs(BACKSPACEEDIT)>

short BACKSPACEEDIT ( short filenum );
```

Syntax for TAL Programmers

```
error := BACKSPACEEDIT ( filenum );          ! i
```

Parameters

error returned value

INT

is a file-system error number indicating the outcome of the operation.

filenum input

INT:value

is the number that identifies the open file on which the operation is to be performed.

Related Programming Manual

For programming information about the BACKSPACEEDIT procedure, see the *Guardian Programmer's Guide*.

BINSEM_CLOSE_ Procedure

Summary

The BINSEM_CLOSE_ procedure closes access to a binary semaphore.

Syntax for C Programmers

```
#include <cextdecs(BINSEM_CLOSE_)>

short BINSEM_CLOSE_ ( __int32_t semid  );
```

Syntax for TAL Programmers

```
error := BINSEM_CLOSE_ ( semid );                    ! i
```

Parameters

error returned value

INT

indicates the outcome of the call:

0 No error.

29 Required parameter missing. The *semid* parameter must be specified.

4022 Invalid parameter. The *semid* parameter does not identify a binary semaphore that is opened by the calling process. The corresponding errno value is ENOENT.

4045 Deadlock. The binary semaphore specified by *semid* cannot be closed because it is locked by the calling process. The corresponding errno value is EDEADLK.

semid input

INT (32):value

specifies a binary semaphore ID.

Considerations

Note. There are additional considerations for privileged callers.

- The close operation and the state of the binary semaphore
  - If the binary semaphore is locked by the calling process, then the value of *error* is 4045, the state of the binary semaphore remains unchanged, and the specified binary semaphore ID can continue to provide access to the binary semaphore.
  - If the binary semaphore is either unlocked, forsaken, or locked by another process, then the value of *error* is 0, the state of the binary semaphore remains unchanged (provided that this is not the last close of the binary

semaphore), and the specified binary semaphore ID can no longer provide access to the binary semaphore.

- When there are no more concurrent opens of the binary semaphore, space used by the binary semaphore is returned to the system main-memory pool.
- A closed binary semaphore ID can be reassigned on subsequent calls to the BINSEM_CREATE_ and BINSEM_OPEN_ procedures.
- For information about binary semaphores for the BINSEM_CREATE_ procedure, see General Considerations for Binary Semaphores on page 2-56.

Example

```
error := BINSEM_CLOSE_( semid );
```

Related Programming Manuals

For programming information about the BINSEM_CLOSE_ procedure, see the *Guardian Programmer's Guide*.

BINSEM_CREATE_ Procedure

Summary

The BINSEM_CREATE_ procedure creates, opens, and locks a binary semaphore.

Syntax for C Programmers

```
#include <cextdecs(BINSEM_CREATE_)>

short BINSEM_CREATE_ ( __int32_t  *semid
                      ,short security );
```

Syntax for TAL Programmers

```
error := BINSEM_CREATE_ ( semid                          ! o
                         ,security );                    ! i
```

Parameters

error returned value

INT

indicates the outcome of the call:

0 No error.

22 Bounds error. The *semid* parameter cannot be written to by the calling process.

29 Required parameter missing. The *semid* and *security* parameters must be specified.

4002 Either the process or the processor has reached the maximum number of user semaphores it can open concurrently. The corresponding errno value is ENOENT.

Note. The number of binary semaphores a processor can have open is 65536. This information is supported only on systems running J06.06 and later J-series RVUs and H06.17 and later H-series RVUs.

4022 Invalid parameter. The *security* parameter is not a valid value. Specifying an invalid value for *security* could cause unpredictable results in future RVUs. The corresponding errno value is EINVAL.

4024 Process cannot open the binary semaphore. The process has reached the maximum number of binary semaphores it can open. The corresponding errno value is EMFILE.

4028 No space. The processor has reached the maximum limit of space available for binary semaphores. The corresponding errno value is ENOSPC.

semid output

INT (32) .EXT:ref:1

returns the binary semaphore ID.

security input

INT:value

specifies the binary semaphore security. The security determines which processes on the same processor can open the binary semaphore. Values are:

0 Any. Any process.

1 Group. Any process with the same process access ID group as the binary semaphore's owner, or a process with the super ID (255,255).

2 Owner. Any process with the same process access ID (group and member) as the binary semaphore's owner, or a process with the super ID.

General Considerations for Binary Semaphores

- Binary semaphore attributes
  - Owner. The owner of a binary semaphore is the process access ID of the process that calls the BINSEM_CREATE_ procedure. The owner of a binary semaphore is defined by a group number and a member number. The owner is relevant only in the context of security; it is neither specified nor returned by the binary semaphore procedures.
  - Security. The security of a binary semaphore determines whether it can be opened by any process, a process belonging to the owner's group, or a process belonging to the owner. Once a binary semaphore has been created, its security cannot be altered.
- Identifying a binary semaphore
  - Binary semaphore ID. A binary semaphore ID identifies each instance of an open of a binary semaphore. A binary semaphore can have multiple openers.
  - Process handle. A process handle is used in conjunction with a binary semaphore ID to identify a binary semaphore opened by another process.
- Three binary semaphore states
  - Locked. A binary semaphore can be locked by a process. Only one process at a time can hold the lock on a binary semaphore.
  - Unlocked. A binary semaphore can have no lock on it.
  - Forsaken. A binary semaphore can be forsaken if it was locked by a process that has terminated.
- Binary semaphore operations

  Operations on a binary semaphores are atomic: they finish one at a time and never finish concurrently. These procedures perform operations on binary semaphores:

- BINSEM_CREATE_ which creates, opens, and locks a binary semaphore. A binary semaphore ID is returned for use with other operations.
- BINSEM_OPEN_ which opens access to a binary semaphore. A binary semaphore ID is returned for use with other operations.
- BINSEM_LOCK_ which locks a binary semaphore.
- BINSEM_UNLOCK_ which unlocks a binary semaphore.
- BINSEM_CLOSE_ which closes access to a binary semaphore.
- BINSEM_FORCELOCK_ which takes the lock from a process that has the lock.
- BINSEM_ISMINE_ which returns whether or not the caller is the current owner of the binary semaphore.

  Note. The BINSEM_ISMINE_procedure is supported on systems running J06.03 and later J-series RVUs and H06.13 and later H-series RVUs.

- Binary semaphore procedures synchronize processes in the same processor

  The binary semaphore procedures cannot be used to synchronize processes on different processors. To synchronize distributed processes, use interprocess messages or file locks as described in the *Guardian Programmer's Guide*.

- Binary semaphore resource requirements

  On systems running H06.16/J06.05 and earlier RVUs, the maximum number of binary semaphores a process can have is 64, and the maximum number of binary semaphores a processor can have open is equal to the number of processes.

  On systems running H06.17/J06.06 and later RVUs, the maximum number of binary semaphores a process can have is 8192, and the maximum number of binary semaphores a processor can have open is 65536.

  In the D30.00 RVU, each open binary semaphore occupies 4 bytes of the process file segment.

Considerations

Note. There are additional considerations for privileged callers.

- The create operation and the state of the binary semaphore

  The binary semaphore is opened and locked by the calling process when the binary semaphore is created.

Example

```
error := BINSEM_CREATE_( semid, security );
```

Related Programming Manuals

For programming information about the BINSEM_CREATE_ procedure, see the *Guardian Programmer's Guide*.

BINSEM_FORCELOCK_ Procedure

Summary

The BINSEM_FORCELOCK_ procedure forces a lock on a binary semaphore. This procedure is used when it is not possible to lock a binary semaphore with the BINSEM_LOCK_ procedure.

Syntax for C Programmers

```
#include <cextdecs(BINSEM_FORCELOCK_)>

short BINSEM_FORCELOCK_ ( __int32_t semid
                        ,short *processhandle );
```

Syntax for TAL Programmers

```
status := BINSEM_FORCELOCK_ ( semid                              !
i
                            ,processhandle );                    !
o
```

Parameters

status returned value

INT

indicates the outcome of the call:

0 No error.

22 Bounds error. The *processhandle* parameter cannot be written to by the calling process.

4022 Invalid parameter. The *semid* parameter does not identify a binary semaphore that is opened by the calling process. The corresponding errno value is EINVAL.

4045 Deadlock. The binary semaphore was forsaken before the procedure call, and it is now locked. The corresponding errno value is EDEADLK.

4103 Already locked. The binary semaphore was locked by the calling process before the procedure call, and it remains locked. The corresponding errno value is EALREADY.

semid input

INT (32):value

specifies the binary semaphore ID.

processhandle output

INT .EXT:ref:10

returns the process handle of the process that previously held the lock on the binary semaphore. A null process handle (-1 in each word) is returned if the binary semaphore was previously unlocked. If *status* is 4045 or 4103, *processhandle* is not updated.

Considerations

Note. There are additional considerations for privileged callers.

- The force-lock operation and the state of the binary semaphore
  - If the binary semaphore is locked by another process, then the value of *status* is 0, the state of the binary semaphore becomes locked by the calling process, and *processhandle* contains the process handle of the process that previously held the lock on the binary semaphore.
  - If the binary semaphore is locked by the calling process, then the value of *status* is 4103, the state of the binary semaphore becomes locked by the calling process, and *processhandle* is not updated.
  - If the binary semaphore is unlocked, then the value of *status* is 0, the state of the binary semaphore becomes locked by the calling process, and *processhandle* contains a null process handle.
  - If the binary semaphore is forsaken, then the value of *status* is 4045, the state of the binary semaphore becomes locked, and *processhandle* is not updated.
- For information about binary semaphores for the BINSEM_CREATE_ procedure, see General Considerations for Binary Semaphores on page 2-56.

Example

```
status := BINSEM_FORCELOCK_( semid, processhandle );
```

Related Programming Manuals

For programming information about the BINSEM_FORCELOCK_ procedure, see the *Guardian Programmer's Guide*.

BINSEM_ISMINE_Procedure

Summary

The BINSEM_ISMINE_procedure allows an application to query whether it is the current owner of any of the semaphores it has obtained access to via a call to either the BINSEM_CREATE_procedure or the BINSEM_OPEN_procedure.

The BINSEM_ISMINE_ procedure passes the semaphore ID, returned by either the BINSEM_CREATE_procedure or the BINSEM_OPEN_ procedure, to `mysemid` and returns either `true` or `false`.

Note. The BINSEM_ISMINE_ procedure returns information without altering the state of the binary semaphore. The BINSEM_ISMINE_ procedure is supported only on systems running H06.13 and later H-series RVUs and J06.03 and later J-series RVUs.

Syntax for C Programmers

```
amlOwner = BINSEM_ISMINE_(mysemid);
```

Syntax for TAL Programmers

```
AMIOWNER :=BINSEM_ISMINE_(MYSEMID);
```

Parameters

mysemid

INT (32):value

identifies the user semaphore from which the application prompts ownership information.

Considerations

- The Ismine operation and the state of the binary semaphore

  If the binary semaphore is locked by the calling process and the `semid` is valid, the BINSEM_ISMINE_ procedure returns a nonzero value.

- For information about binary semaphores for the BINSEM_CREATE_ procedure, see General Considerations for Binary Semaphores on page 2-56.

Example

```
AMIOWNER:=BINSEM_ISMINE_(MYSEMID);
```

BINSEM_LOCK_ Procedure

Summary

The BINSEM_LOCK_ procedure locks a binary semaphore.

Syntax for C Programmers

```
#include <cextdecs(BINSEM_LOCK_)>

short BINSEM_LOCK_ ( __int32_t semid
                    ,__int32_t timeout );
```

Syntax for TAL Programmers

```
status := BINSEM_LOCK_ ( semid                                  ! i
                        ,timeout );                             ! i
```

Parameters

status returned value

INT

indicates the outcome of the call:

0	No error. The binary semaphore becomes locked.
4011	Operation timed out. The *timeout* value was reached before the binary semaphore could be locked. The corresponding errno value is EAGAIN.
4022	Invalid parameter. The *semid* parameter does not identify a binary semaphore that is opened by the calling process. The corresponding errno value is EINVAL.
4045	Deadlock. The binary semaphore was forsaken before the procedure call, and it is now locked. The corresponding errno value is EDEADLK.

semid input

INT (32):value

specifies the binary semaphore ID.

timeout input

INT (32):value

specifies how many hundredths of a second the procedure should wait for the binary semaphore to become unlocked. The maximum value is 2,147,483,647. A value of -1D causes the procedure to wait indefinitely. A value of 0D causes the procedure to return immediately to the calling process, regardless of whether the binary semaphore is locked.

Considerations

Note. There are additional considerations for privileged callers.

- The lock operation and the state of the binary semaphore
  - If the binary semaphore is unlocked before the specified timeout has elapsed, then the value of *status* is 0 and the state of the binary semaphore becomes locked by the calling process.
  - If the binary semaphore is forsaken before the timeout specified, then the value of *status* is 4045 and the state of the binary semaphore becomes locked by the calling process.
  - If the binary semaphore is locked by another process or by the calling process and the timeout expires, then the value of *status* is 4011 and the state of the binary semaphore remains unchanged.
- Locking a binary semaphore

  If the calling process terminates during the lock operation, the state of the binary semaphore is not changed.

  The same process that locks a binary semaphore should also unlock it.

  A binary semaphore locked by a process that has terminated becomes forsaken. Any process that waits on binary semaphores must account for the possibility of a forsaken binary semaphore. The binary semaphore procedures allow for recovery of a binary semaphore from the forsaken state to the locked state.

  Applications should account for a deadlock condition. For example, a deadlock can occur if two processes require the locks on two binary semaphores and each process holds the lock on one of the binary semaphores. A method of avoiding deadlock situations is to lock binary semaphores in a predetermined order.

  The lock operation finishes in any order regardless of when a process requests the lock or the process priority. So, a lower-priority process could get the lock and lock out a higher-priority process even though the higher-priority process requests it first.
- Searching for processes waiting for a lock

  Call PROCESS_GETINFOLIST_ with the process list attribute (attribute code 15, bit <7>) to determine whether a process is on the binary semaphore list waiting to lock a binary semaphore.

  Note. This information is applicable only for G-series RVUs.
- For information about binary semaphores for the BINSEM_CREATE_ procedure, see General Considerations for Binary Semaphores on page 2-56.

Example

```
status := BINSEM_LOCK_( semid, timeout );
```

BINSEM_OPEN_ Procedure

Summary

The BINSEM_OPEN_ procedure opens a binary semaphore.

Syntax for C Programmers

```
#include <cextdecs(BINSEM_OPEN_)>

short BINSEM_OPEN_ ( __int32_t *semid
                    ,short *processhandle
                    ,__int32_t proc-semid );
```

Syntax for TAL Programmers

```
error := BINSEM_OPEN_ ( semid                          ! o
                       ,processhandle                  ! i
                       ,proc-semid );                  ! i
```

Parameters

error returned value

INT

indicates the outcome of the call:

0	No error.
22	Bounds error. *The semid* parameter cannot be written by the calling process or *processhandle* cannot be read from the calling process.
29	Required parameter missing. The *semid*, *processhandle*, and *proc-semid* parameters must be specified.

4002 Either the process or the processor has reached the maximum number of user semaphores it can open concurrently. The corresponding `errno` value is `ENOENT`.

Note. The number of binary semaphores a processor can have open is 65536. This information is supported only on systems running J06.06 and later J-series RVUs and H06.17 and later H-series RVUs.

4013 Invalid access. The calling process does not have access to the binary semaphore because of its security. The security for a binary semaphore is set by the BINSEM_CREATE_ procedure. The corresponding `errno` value is `EACCES`.

4022 Invalid parameter. The *`processhandle`* parameter does not specify a process. A process that is being created or is terminating is treated as though it does not exist. The corresponding `errno` value is `EINVAL`.

4024 Process cannot open the binary semaphore. The process has reached the maximum limit of binary semaphores it can open. The corresponding `errno` value is `EMFILE`.

`semid` output

INT (32) .EXT:ref:1

returns the binary semaphore ID of this instance of the open binary semaphore. The *`semid`* parameter might have the same value as the *`proc-semid`* parameter.

`processhandle` input

INT .EXT:ref:10

specifies the process handle of a process that has already opened the binary semaphore. The *`processhandle`* and *`proc-semid`* pair can be obtained by a previous call to the BINSEM_CREATE_ or BINSEM_OPEN_ procedure.

`proc-semid` input

INT (32):value

specifies the binary semaphore ID of an open binary semaphore. The *`processhandle`* and *`proc-semid`* pair can be obtained by a previous call to the BINSEM_CREATE_ or BINSEM_OPEN_ procedure.

Considerations

Note. There are additional considerations for privileged callers.

- The open operation and the state of the binary semaphore

  The state of the binary semaphore remains unchanged from its original state.

- For information about binary semaphores for the BINSEM_CREATE_ procedure, see General Considerations for Binary Semaphores on page 2-56.

Example

```
error := BINSEM_OPEN_( semid, processhandle, proc^semid );
```

Related Programming Manuals

For programming information about the BINSEM_OPEN_ procedure, see the *Guardian Programmer's Guide*.

BINSEM_UNLOCK_ Procedure

Summary

The BINSEM_UNLOCK_ procedure unlocks a binary semaphore.

Syntax for C Programmers

```
#include <cextdecs(BINSEM_UNLOCK_)>

short BINSEM_UNLOCK_ ( __int32_t semid  );
```

Syntax for TAL Programmers

```
error := BINSEM_UNLOCK_ ( semid );                    ! i
```

Parameters

error returned value

INT

indicates the outcome of the call:

0 No error.

4001 Cannot unlock. The *semid* parameter is not locked by the calling process. The corresponding errno value is EPERM.

4022 Invalid parameter. The *semid* parameter does not identify a binary semaphore that is opened by the calling process. The corresponding errno value is EINVAL.

semid input

INT (32):value

specifies a binary semaphore ID.

Considerations

- The unlock operation and the state of the binary semaphore
  - If the binary semaphore is locked by the calling process, then the value of *error* is 0 and the state of the binary semaphore becomes unlocked.
  - If the binary semaphore is either locked by another process, unlocked, or forsaken, then the value of *error* is 4001 and the state of the binary semaphore remains unchanged.
- For information about binary semaphores for the BINSEM_CREATE_ procedure, see General Considerations for Binary Semaphores on page 2-56.

Example

```
error := BINSEM_UNLOCK_( semid );
```

Related Programming Manuals

For programming information about the BINSEM_UNLOCK_ procedure, see the *Guardian Programmer's Guide*.

BREAKMESSAGE_SEND_ Procedure

Summary

The BREAKMESSAGE_SEND_ procedure sends a break-on-device message to a specified process.

Syntax for C Programmers

```
#include <cextdecs(BREAKMESSAGE_SEND_)>

short BREAKMESSAGE_SEND_ ( short *processhandle
                          ,short receiver-filenum
                          ,[ short *breaktag ] );
```

Syntax for TAL Programmers

```
error := BREAKMESSAGE_SEND_ ( processhandle      ! i
                             ,receiver-filenum   ! i
                             ,[ breaktag ] );    ! i
```

Parameters

error returned value

INT

returns a file-system error number indicating the outcome of the operation. A successful indication implies only that the message has been sent, not that it has been received.

processhandle input

INT .EXT:ref:10

specifies the process handle of the process that is to receive the break-on-device message.

receiver-filenum input

INT:value

specifies the file number by which the receiving process identifies the open of the process that is sending the break-on-device message.

breaktag input

INT .EXT:ref:2

if present, specifies a user-defined value to be delivered in the break-on-device message. This value corresponds to the break tag value that can be supplied to an access method with SETPARAM function 3.

If this parameter is omitted, 0 is used.

Considerations

If *processhandle* designates a member of a named process pair, and if a failure or a path switch occurs, delivery of the break-on-device message is automatically retried to the backup process. For detailed information about system messages, see the *Guardian Procedure Errors and Messages Manual*.

Example

```
error := BREAKMESSAGE_SEND_( proc-handle, file-number, tag );
```

Related Programming Manual

For programming information about the BREAKMESSAGE_SEND_ procedure, see the *Guardian Programmer's Guide*.

3 Guardian Procedure Calls (C)

This section contains detailed reference information for all user-accessible Guardian procedure calls beginning with the letter C. Table 3-1 lists all the procedures in this section.

Table 3-1. Procedures Beginning With the Letter C (page 1 of 2)

Table 3-1. Procedures Beginning With the Letter C (page 2 of 2)

CANCEL Procedure

Summary

The CANCEL procedure is used to cancel the oldest incomplete operation on a file opened nowait. The canceled operation might or might not have had effects. For disk files, the file position might or might not be changed.

Note. You can cancel a specific request, identified with a *tag* parameter, using a call to CANCELREQ.

Syntax for C Programmers

```
#include <cextdecs(CANCEL)>

_cc_status CANCEL ( short filenum );
```

- The function value returned by CANCEL, which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL CANCEL ( filenum );                    ! i
```

Parameters

filenum input

INT:value

is the number of an open file whose oldest incomplete operation you want to cancel.

Considerations

- Queue files

  If a READUPDATELOCK[X] operation is canceled using the CANCEL procedure, the READUPDATELOCK[X] might already have deleted a record from the queue file, which could result in a loss of a record from the queue file. For audited queue files only, your application can recover from a timeout error by calling the ABORTTRANSACTION procedure, when detecting error 40, to ensure that any dequeued records are reinserted into the file. For nonaudited queue files, there is no recovery of a lost record. Thus, your application should never call AWAITIO[X] with a time limit greater than 0D if READUPDATELOCK[X] is pending. The ABORTTRANSACTION recovery procedure does not work on nonaudited queue files.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILE_GETINFO_ or FILEINFO).

= (CCE) indicates that the operation was canceled.

> (CCG) does not return from CANCEL.

Messages

The server process (that is, a process that was opened and to which the I/O request was sent) receives a system message -38 (queued message cancellation) that identifies the canceled I/O request, if it has requested receipt of such messages. If the server has already replied to the I/O request, message -38 is not delivered. For details about system message-38, see the *Guardian Procedure Errors and Messages Manual*.

Related Programming Manual

For programming information about the CANCEL procedure, see the *Guardian Programmer's Guide*.

CANCELPROCESSTIMEOUT Procedure

Summary

The CANCELPROCESSTIMEOUT procedure cancels a process-time timer previously initiated by a call to the SIGNALPROCESSTIMEOUT procedure.

Syntax for C Programmers

```
#include <cextdecs(CANCELPROCESSTIMEOUT)>

_cc_status CANCELPROCESSTIMEOUT ( short tag );
```

- The function value returned by CANCELPROCESSTIMEOUT, which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL CANCELPROCESSTIMEOUT ( tag );              ! i
```

Parameters

tag input

INT:value

is the identifier associated with the timer to be canceled or -1 if all timers set by calls to SIGNALPROCESSTIMEOUT by that process are to be canceled.

Condition Code Settings

< (CCL) is not returned by CANCELPROCESSTIMEOUT.

= (CCE) indicates that CANCELPROCESSTIMEOUT was successful.

> (CCG) indicates that *tag* was invalid.

Related Programming Manual

For programming information about the CANCELPROCESSTIMEOUT procedure, see the *Guardian Programmer's Guide*.

CANCELREQ[L] Procedure

Summary

The CANCELREQ[L] procedure is used to cancel an incomplete operation, identified by a file number and tag, on a file opened for nowait I/O. Use the CANCELREQL procedure to cancel an outstanding SERVERCLASS_SENDL_ request, which has a 64-bit tag. The canceled operation might or might not have had effects. For disk files, the file position might or might not be changed.

Syntax for C Programmers

```
#include <cextdecs(CANCELREQ)>

_cc_status CANCELREQ ( short filenum
                     ,[ __int32_t tag ] );


#include <cextdecs(CANCELREQL)>

short CANCELREQL ( short filenum
                 ,[ long long tag ] );
```

- CEXTDECS (through the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.
- The function value returned by CANCELREQ, which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL CANCELREQ ( filenum                        ! i
                ,[ tag ] );                     ! i


error:= CANCELREQL ( filenum                    ! i
                    ,[ tag ] );                 ! i
```

Parameters

error returned value

INT:value (Use with CANCELREQL)

is a file-system error number indicating the outcome of the operation.

0 (FEOK)

indicates a successful operation.

filenum input

INT:value

is the number of an open file, identifying the file whose operation did not complete and is to be canceled.

tag input

INT(32):value (Use with CANCELREQ)
INT(64):value (Use with CANCELREQL)

is the tag value passed to the procedure that initialized the I/O operation. It identifies the operation to be canceled.

If the *tag* omitted or 0, the oldest incomplete request is canceled.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILE_GETINFO_ or FILEINFO).

= (CCE) indicates that the operation was canceled.

> (CCG) does not return from CANCELREQ.

Considerations

- If you use the *tag* parameter, the system cancels the oldest incomplete operation associated with that tag value. If you do not provide a *tag*, the system cancels the oldest incomplete operation for *filenum*.
- If you omit the *tag* parameter, CANCELREQ[L] works exactly like CANCEL.

Messages

The server process receives a system message -38 (queued message cancellation) that identifies the canceled I/O request, if it has requested receipt of such messages.

Related Programming Manual

For programming information about the CANCELREQ procedure, see the *Guardian Programmer's Guide*.

Note. The CANCELREQL procedure is supported on systems running J06.07 and later J-series RVUs and H06.18 and later H-series RVUs.

CANCELTIMEOUT Procedure

Summary

The CANCELTIMEOUT procedure cancels an elapsed-time timer previously initiated by a call to the SIGNALTIMEOUT procedure.

Syntax for C Programmers

```
#include <cextdecs(CANCELTIMEOUT)>

_cc_status CANCELTIMEOUT ( short tag );
```

- The function value returned by CANCELTIMEOUT, which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL CANCELTIMEOUT ( tag );              ! i
```

Parameters

tag input

INT:value

is the identifier associated with the timer to be canceled or -1 if all timers set by calls to SIGNALTIMEOUT by that process are to be canceled.

Condition Code Settings

< (CCL) is not returned from CANCELTIMEOUT.

= (CCE) indicates that CANCELTIMEOUT completed successfully.

> (CCG) indicates that *tag* was invalid.

Related Programming Manual

For programming information about the CANCELTIMEOUT procedure, see the *Guardian Programmer's Guide*.

CHANGELIST Procedure

Summary

The CHANGELIST procedure is used only when the application program acts as a supervisor or tributary station in a centralized multipoint configuration.

Within a supervisor station, CHANGELIST performs one of these operations:

- Specifies continuous or noncontinuous polling
- Enables or disables polling of a particular station
- Resumes polling of partially disabled (that is, nonresponding) stations
- Performs the activation or deactivation of a tributary station by altering the setting of the poll state bit for a particular entry

---

Note. If polling is in progress when you make the call to CHANGELIST, the specified changes do not take effect until polling completes either on its own or as the result of a call to HALTPOLL.

Syntax for C Programmers

```
#include <cextdecs(CHANGELIST)>

_cc_status CHANGELIST ( short filenum
                       ,short function
                       ,short parameter );
```

- The function value returned by CHANGELIST, which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL CHANGELIST ( filenum                          ! i
                 ,function                         ! i
                 ,parameter );                     ! i
```

Parameters

filenum input

INT:value

is the one-word integer variable returned by the call to FILE_OPEN_ or OPEN that opened the line.

function input

INT:value

is an integer value specifying what change is to be made:

>= 0 changes the poll state bit. In this case, *function* also specifies the relative address of the particular station address within the address list (0 indicates the first entry, 1 the second entry, and so forth). The *parameter* value, described below, specifies whether you want the bit to be set or cleared.

-1 changes the polling type. The *parameter* value described below specifies whether you want continuous polling or you want the polling list to be traversed a finite number of times.

-2 restores all partially disabled stations.

parameter input

INT:value

is an integer value used with the *function* value to specify what change is to be made.

>= 0 The *parameter* value specifies whether you want the poll or select state bit set or cleared as follows:

0 Cleared

1 Set

The meaning of this bit is somewhat different depending upon whether the station list is that of a supervisor or a tributary station:

- Within a supervisor station, the poll state bit enables (clears) or disables (sets) the polling of the particular tributary station.
- Within a tributary station, the poll state bit activates (clears) or deactivates (sets) the tributary station with regard to its ability to respond to a poll or select the designated station address.

= -1 The *parameter* value specifies the desired type of polling as follows:

0 Continuous polling

> 0 Noncontinuous polling (traverse the polling list the specified number of times and then cease polling).

= -2 The *parameter* value has no meaning. The CHANGELIST procedure, however, expects to be passed three values; you must therefore supply a dummy *parameter* value.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILE_GETINFO_ or FILEINFO).

= (CCE) indicates that the CHANGELIST procedure executed successfully.

> (CCG) does not return from CHANGELIST.

Examples

```
CALL CHANGELIST ( FNUM , -1 , 10 );
```

In this example, within a supervisor station, this call enables limited polling in which the station list is traversed 10 times. Polling does not begin, however, until READ is subsequently called. After the tenth pass through the polling list, polling ceases.

Related Programming Manuals

For programming information about the CHANGELIST procedure, see the data communication manuals.

CHECK^BREAK Procedure

Summary

The CHECK^BREAK procedure tests whether the BREAK key has been typed since the last CHECK^BREAK.

CHECK^BREAK is a sequential I/O (SIO) procedure and can be used only with files that have been opened by OPEN^FILE.

Syntax for C Programmers

```
#include <cextdecs(CHECK_BREAK)>

short CHECK_BREAK ( short { _near *common-fcb }
                          { _near *file-fcb   } );
```

Sytax for TAL Programmers

```
state := CHECK^BREAK ( { common-fcb }            ! i
                       { file-fcb   } );         ! i
```

Parameters

state returned value

INT

returns a value indicating whether or not the BREAK key has been typed. Values are:

1 BREAK key typed; the process owns BREAK.

0 BREAK key not typed; this process does not own BREAK.

common-fcb input

INT:ref:*

identifies the file to be checked for BREAK. The *common-Fcb* parameter is allowed for convenience.

file-fcb input

INT:ref:*

identifies the file to be checked for BREAK.

Considerations

- Default action

  If a carriage return/line feed (CR/LF) on BREAK is enabled (that is, BREAK ownership is taken by the process), the CR/LF default case sequence is executed on the terminal where BREAK is typed.

- For information about terminals, see the *Guardian Programmer's Guide*.

Example

```
BREAK := CHECK^BREAK ( OUT^FILE );
```

Related Programming Manual

For programming information about the CHECK^BREAK procedure, see the *Guardian Programmer's Guide*.

CHECK^FILE Procedure

Summary

The CHECK^FILE procedure checks the file characteristics of a specified file.

CHECK^FILE is a sequential I/O (SIO) procedure and can be used only with files that have been opened by OPEN^FILE.

Syntax for Native C Programs

```
#include <cextdecs(CHECK_FILE)>

short CHECK_FILE ( short { _near *common-fcb }
                         { _near *file-fcb   }
                  ,short operation
                  ,[ short _near *ret-addr ] );
```

Syntax for TNS C Programs

```
#include <cextdecs(CHECK_FILE)>

short CHECK_FILE ( short { _near *common-fcb }
                         { _near *file-fcb   }
                  ,short operation );
```

Syntax for TAL Programmers

For pTAL callers, the procedure definition is:

```
retval := CHECK^FILE ( { common-fcb }              ! i
                       { file-fcb   }              ! i
                        ,operation                 ! i
                        ,[ ret-addr ] );           ! o
```

For other callers, the procedure definition is:

```
retval := CHECK^FILE ( { common-fcb }              ! i
                       { file-fcb   }              ! i
                        ,operation );              ! i
```

Parameters

retval returned value

INT

returns a value for the requested operation. The operations and their associated return values are listed in Table 3-2 and Table 3-3.

common-fcb or *file-fcb* input

INT:ref:*

identifies which file is checked. The common file control block (FCB) can be used for certain types of operations; the common FCB must be used for the operations FILE^BREAKHIT, FILE^ERRORFILE, and FILE^TRACEBACK. Specifying an improper FCB causes an error indication.

operation input

INT:value

specifies which file characteristic is checked. The operations and their associated return values are listed in Table 3-2 and Table 3-3.

ret-addr output

WADDR

for native callers only, returns an address for the requested operation.

Considerations

- During the execution of this procedure, the detection of any error causes the display of an error message and the process is aborted.
- This procedure is used to get the primary extent or secondary extent size of a file that is no greater than 65,535 pages. If the primary and secondary extent sizes are greater than 65,535, error 538 is returned.
- Table 3-2 contains operations that return values returned in *retval*.
- Table 3-3 contains operations that return addresses. For native callers, addresses are returned in the *ret-addr* parameter. For other callers, addresses are returned in *retval*.
- In Table 3-2 and Table 3-3 the column labeled “State of File” can contain these:

  Open The file must be open to obtain this characteristic.

  Any The file can be either open or closed.

Table 3-2 describes operations that return values returned in *retval*.

Table 3-2. CHECK^FILE Operations That Return Values (page 1 of 4)

operation	State of File	*retval*
FILE^ABORT^XFERERR	Open	0 if the process is not to abort upon detection of a fatal error in the file. 1 if the process is to abort.
FILE^ASSIGNMASK1	Any	Returns the high-order word of the ASSIGN message field mask in the FCB. This value generally has meaning only after being set by the INITIALIZER procedure.
FILE^ASSIGNMASK2	Any	Returns the low-order word of the ASSIGN message field mask in the FCB. This value generally has meaning only after being set by the INITIALIZER procedure.
FILE^BLOCKBUFLEN	Any	Returns a count of the number of bytes used for blocking.
FILE^BREAKHIT	Any	0 if the break hit bit is equal to 0 in the FCB. 1 if the break hit bit is equal to 1 in the FCB. The break hit bit is an internal indicator normally used only by the SIO procedures. Note: When using the break-handling procedures, do not use FILE^BREAKHIT to determine whether the BREAK key has been pressed. Instead, the CHECK^BREAK procedure must be called.
FILE^CHECKSUM	Any	Returns the value of the checksum word in the FCB.
FILE^COUNTXFERRED	Open	Returns a count of the number of bytes transferred in the latest physical I/O operation.
FILE^CREATED	Open	0 if a file was not created by OPEN^FILE. 1 if a file was created by OPEN^FILE.
FILE^CRLF^BREAK	Open	0 if no CR/LF sequence is to be issued to the terminal upon break detection. 1 if this sequence is to be issued.

Table 3-2. CHECK^FILE Operations That Return Values (page 2 of 4)

operation	State of File	*retval*
FILE^EDITLINE^INCREMENT	Open	Returns the EDIT line increment to be added to successive line numbers for lines that are added to the file. The value is 1000 times the line number increment value.
FILE^ERROR	Any	Returns the error number of the latest error that occurred within the file.
FILE^FILEFORMAT	Open	Returns the file format type. Returns 1 for format 1 files and 2 for format 2 files.
FILE^FILEINFO	Open	<file-info>, where <file-info>.<0:3> = File type: 0 = Unstructured 1 = Relative 2 = Entry-sequenced 3 = Key-sequenced 4 = EDIT 8 = Odd unstructured .<4:9> = Device type .<10:15>= Device subtype The device type and subtype are described in Appendix A, Device Types and Subtypes. File types 0-3 are described in the *Enscribe Programmer's Guide*.
FILE^FNUM	Open	Returns the file number. If the file is not open, the file number is -1.
FILE^LEVEL3^SPOOLING	Open	0 if level-3 spooling is disabled. 1 if level-3 spooling is enabled. See OPEN^FILE for details.
FILE^LOGIOOUT	Open	0 if there is no logical I/O outstanding. 1 if a logical read is outstanding. 2 if a logical write is outstanding.
FILE^OPENACCESS	Any	Returns the open access for the file. See SET^FILE for the format.
FILE^OPENEXCLUSION	Open	Returns the open exclusion for the file. See SET^FILE for format.

Table 3-2. CHECK^FILE Operations That Return Values (page 3 of 4)

operation	State of File	*retval*
FILE^PHYSIOOUT	Open	0 to indicate that there is no outstanding physical I/O operation. 1 if a physical I/O operation is outstanding.
FILE^PRIEXT	Any	Returns the file's primary extent size in pages. The primary extent size cannot be greater than 65,535 pages. Otherwise, error 538 is returned.
FILE^PRINT^ERR^MSG	Open	0 if no error message is to be printed upon detection of a fatal error in the file. 1 if an error message is to be printed.
FILE^PROMPT	Open	Returns the interactive prompt character for the file in <9:15>.
FILE^RCVEOF	Open	0 if the user does not get an end-of-file (EOF) indication when the process [pair] having this process open closes it. 1 if the user does get an EOF indication when this process closes.
FILE^RCVOPENCNT	Open	Returns a count of current openers of this process {0:2}. At any given moment, openers are limited to a single process [pair].
FILE^RCVUSEROPENREPLY	Open	0 if the SIO procedures are to reply to the open messages ($RECEIVE file). 1 if the user is to reply to the open messages.
FILE^READ^TRIM	Open	0 if the trailing blanks are not trimmed off the data read from this file. 1 if the trailing blanks are trimmed.
FILE^RECORDLEN	Any	Returns the logical record length.
FILE^SECEXT	Any	Returns the file's secondary extent size in pages. The secondary extent size cannot be greater than 65,535 pages. Otherwise, error 538 is returned.

Table 3-2. CHECK^FILE Operations That Return Values (page 4 of 4)

operation	State of File	*retval*
FILE^SYSTEMMESSAGES	Open	Returns a mask word indicating which system messages the user handles directly. See SET^FILE for the format. 0 indicates that the SIO procedures handle all system messages. Note that this operation cannot check some of the newer system messages; for these, use operation FILE^SYSTEMMESSAGESMANY.
FILE ^SYSTEMMESSAGESMANY	Open	Returns the word address within the FCB of a four-word mask indicating which system messages the user handles directly. See SET^FILE for the format. A return of all zeros indicates that the SIO procedures handle all system messages.
FILE^TRACEBACK	Any	0 if the P-relative address should not be appended to all SIO error messages. 1 if the P-relative address should be appended to all SIO error messages.
FILE^USERFLAG	Any	Returns the user flag word. (See SET^FLAG procedure, SET^USERFLAG operation.)
FILE^WRITE^FOLD	Open	0 if records longer than the logical record length are truncated. 1 if long records are folded.
FILE^WRITE^PAD	Open	0 if a record shorter than the logical record length is not padded with trailing blanks before it is written to the file. 1 if a short record is padded with trailing blanks.
FILE^WRITE^TRIM	Open	0 if trailing blanks are not trimmed from data written to the file. 1 if trailing blanks are trimmed.

This table describes operations that return addresses. For native callers, addresses are returned in the *ret-addr* parameter. For other callers, addresses are returned in *retval*.

Table 3-3. CHECK^FILE Operations That Return Addresses (page 1 of 2)

operation	**State of File**	*ret-addr* (for native callers) *retval* (for other callers)
FILE^BWDLINKFCB	Any	Returns the address of the FCB pointed to by the backward link pointer within the FCB. This indicates the linked-to FCBs that need to be checkpointed after an OPEN^FILE or CLOSE^FILE call.
FILE^DUPFILE	Open	Returns the word address of the duplicate file FCB. 0 is returned if there is no duplicate file.
FILE^ERROR^ADDR	Any	Returns the word address within the FCB where the error code is stored.
FILE^ERRORFILE	Any	Returns the word address within the FCB of the reporting error file. 0 is returned if there is none.
FILE^FCB^ADDR	Any	Returns the address of the FCB.
FILE^FILENAME^ADDR	Any	Returns the word address within the FCB of the physical file name.
FILE^FNUM^ADDR	Any	Returns the word address within the FCB of the file number.
FILE^FWDLINKFCB	Any	Returns the address of the FCB pointed to by the forward link pointer within the FCB. This value indicates the linked-to FCBs that need to be checkpointed after an OPEN^FILE or CLOSE^FILE call.
FILE^LOGICALFILENAME^ADDR	Any	Returns the word address within the FCB of the logical file name. The logical file name is encoded as follows: Byte Number — Contents [0] — <len> is the length of the logical file name in bytes {0:8} [1] through [8] — <logical file name>

Table 3-3. CHECK^FILE Operations That Return Addresses (page 2 of 2)

operation	**State of File**	*ret-addr* (for native callers) *retval* (for other callers)
FILE^OPENERSPID^ADDR	Open	Returns the word address within the FCB of the file opener's PID. Valid only for C-series format FCBs.
FILE^SEQNUM^ADDR	Any	Returns the word address within the FCB of an INT (32) sequence number. This is the line number of the last record read of an EDIT file. For other files, this is the sequence number of the last record read multiplied by 1000.
FILE^USERFLAG^ADDR	Any	Returns the word address within the FCB of the user flag word.

Example

native caller:

```
CALL CHECK^FILE (IN^FILE , FILE^FILENAME^ADDR, INFILE^ADDR);
```

other callers:

```
@INFILE^NAME := CHECK^FILE (IN^FILE , FILE^FILENAME^ADDR);
```

Related Programming Manual

For programming information about the CHECK^FILE procedure, see the *Guardian Programmer's Guide*.

CHECKALLOCATESEGMENT Procedure (Superseded by SEGMENT_ALLOCATE_CHKPT_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The CHECKALLOCATESEGMENT procedure allocates a selectable extended data segment for use by the backup process in a process pair. It is called from the primary process.

Although it is possible to share flat segments meant for use by the backup process in a process pair using the CHECKALLOCATESEGMENT procedure, flat segments can be allocated for this purpose only with the SEGMENT_ALLOCATE_CHKPT_ procedure. SEGMENT_ALLOCATE_CHKPT_ can also allocate selectable segments for use by the backup process in a process pair.

Syntax for C Programmers

This passive backup procedure is not supported in C programs. For a comparison of active backup and passive backup, see the *Guardian Programmer's Guide*.

Syntax for TAL Programmers

```
CALL CHECKALLOCATESEGMENT ( segment-id                  ! i
                          ,[ file-name ]                ! i
                          ,[ pin-and-flags ]            ! i
                          ,error );                     ! o
```

Parameters

segment-id input

INT:value

is the number by which the process chooses to refer to the extended data segment. Segment IDs are in these ranges:

0-1023 can be specified by user processes.

Other IDs are reserved for HP software.

No process can supply a segment ID greater than 2047.

file-name input

INT .EXT:ref:12

if present, is the internal-format file name of a swap file to be associated with the extended data segment. If the file exists, all data in the file is used as initial data for the segment. If the file does not exist, one is created. Remote file names and structured files are not accepted. If the process terminates without deallocating the segment, any data still in memory is written back out to the file. CHECKALLOCATESEGMENT must be able to allocate a sufficient number of file extents to contain all memory in the segment.

The parameter can be a volume name with a blank subvolume and file; CHECKALLOCATESEGMENT allocates a temporary swap file on the indicated volume.

If you do not specify *file-name* (and if a segment is not being shared using the PIN method), CHECKALLOCATESEGMENT uses the volume of the data stack swap file to create a temporary swap file for the new segment.

pin-and-flags input

INT:value

Defaults to %040000. Its values are:

<8:15> Optional PIN for segment sharing.

Requests allocation of a shared segment that is shared by the PIN method. This value specifies the process identification number (PIN) of the process that has previously allocated the segment and with which the caller wants to share the segment. This value is not used if bit 1 is set to 1 (see bit 1 later).

<5:7> Not used; must be zero (0).

<4> If 1, requests allocation of an extensible segment. An extensible segment is an extended data segment for which the underlying swap file disk space is not allocated until needed. In this case, the value of *segment-size* allocated by the primary process is taken as a

maximum size, and the underlying virtual memory is expanded dynamically as the user accesses various addresses within the extended data segment. When the user first accesses a portion of an extensible data segment for which the corresponding swap file extent hasn't been allocated, the operating system allocates the extent. If this extent cannot be allocated, the user process terminates: a TNS Guardian process terminates with a "no memory available" trap (trap 12); an OSS or native process receives a SIGNOMEM signal.

pin-and-flags (continued)

<3> If 1, requests allocation of a "shared segment." A shared segment is an extended data segment that can be shared with other processes in the processor. The *file-name* parameter must be supplied when a shared segment is allocated. Processes sharing segments by this mechanism can reference the address space by different segment IDs and can supply different values of *segment-size* to ALLOCATESEGMENT. The value of *segment-size* supplied by the very first allocator of a particular shared segment (as identified by the swap file name) limits the size of the segment for subsequent processes attempting to share that segment.

<2> If 1, requests allocation of a read-only segment. A read-only segment is an extended data segment that is initialized from a preexisting swap file and used only for read access. A read-only segment can be shared by either the PIN or file-name method. It can also be shared by file name between processes in different processors. Note that the *file-name* parameter must specify the name of an existing swap file that is not empty. If this bit is 1, bit <4> of *pin-and-flags* must be 0 (writeback-inhibit extensible segments are not allowed) and bit 1 must be set to 1, indicating a shared segment.

<1> If 1, bits <8:15> are ignored.

If 0, designates that the extended data segment specified by *segment-id* is to be shared with the process specified by the PIN in bits <8:15> of *pin-and-flags*. For this sharing to occur, the processes must execute in the same processor and one of these must be true:

- The processes share the same process access ID (PAID).
- This process's PAID must be the group manager for the PAID of the other process.
- This process's PAID must be the super ID (255,255).

Processes sharing a segment by the PIN method reference the segment by the same value of *segment-id*.

error output

INT .EXT:ref:1

indicates the outcome of the call. This procedure returns all values returned by ALLOCATESEGMENT in the backup, plus these file-system errors:

2	Segment is not allocated by the primary, or segment ID is invalid
22	Bounds error on file name
29	The *segment-id* is missing
30	No message-system control blocks available
31	Cannot use the PFS, or there is no room in the PFS for a message buffer in either the backup or the primary
201	Unable to LINK to the backup

Condition Code Settings

< (CCL)	is set if the *error* parameter is missing, or there is a bounds error on the *error* parameter.
= (CCE)	is set by all other errors (see the *error* parameter).
> (CCG)	is never returned from this procedure.

Considerations

- The *segment-size* parameter of ALLOCATESEGMENT is not supported because the size of the primary process's segment is used.
- An extended data segment with the same segment ID must be previously allocated in the primary process; that is, before the call to CHECKALLOCATESEGMENT.
- If the *file-name* parameter is provided, that file name is used by ALLOCATESEGMENT in the backup process; otherwise, no file-name parameter is passed to ALLOCATESEGMENT in the backup process.
- If the *pin-and-flags* parameter is omitted, the default value is used (%040000).
- Be careful when using the *pin-and-flags* parameter. The flag settings should be the same as the flag settings used when the extended data segment was allocated by the primary process. CHECKALLOCATESEGMENT does not check the flag settings; the information is no longer available.

  If a PIN is specified, assign it carefully, because the PIN may not necessarily be the same on the backup processor. You must determine the correct PIN for the backup processor.
- If the extended data segment is not read-only, the swap file name must be different on the backup and primary processors because swap files cannot be shared

between processors. An error is returned from ALLOCATESEGMENT on the backup.

- Nonexisting temporary swap file

  If a shared segment is being allocated (*pin-and-flags* bits <3:2> not equal to 0), and a volume name only is supplied in the *file-name* parameter, then the complete file name of the temporary file created by CHECKALLOCATESEGMENT is returned.

- Swap file extent allocation

  If an extensible segment is being created, then only one extent of the swap file is allocated when CHECKALLOCATESEGMENT returns.

- Segment sharing

  Subject to security requirements, a process can share a segment with another process running on the same processor. For example, process $X can share a segment with any of these processes on the same processor:

  - Any process that has the same process access ID (PAID)
  - Any process that has the same group ID, if $X is the group manager (n,255)
  - Any process, if $X is the super ID (255,255)

  If processes are running in different processors, they can share a segment only if the security requirements are met and the segment is a read-only segment.

  Callers of [CHECK]ALLOCATESEGMENT can share segments with callers of SEGMENT_ALLOCATE_[CHKPT_]. High-PIN callers can share segments with low-PIN callers.

- Sharing flat segments

  A process cannot share a flat segment with a process that allocated a selectable segment, because the segments reside in different parts of memory. (Similarly, a process cannot share a selectable segment with a process that allocated a flat segment.)

CHECKCLOSE Procedure (Superseded by FILE_CLOSE_CHKPT_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The CHECKCLOSE procedure is called by a primary process to close a designated file in its backup process.

The backup process must be in the monitor state (that is, in a call to CHECKMONITOR) for the CHECKCLOSE to be successful. The call to CHECKCLOSE causes the CHECKMONITOR procedure in the backup process to call the file-system CLOSE procedure for the designated file.

Syntax for C Programmers

This passive backup procedure is not supported in C programs. For a comparison of active backup and passive backup, see the *Guardian Programmer's Guide*.

Syntax for TAL Programmers

```
CALL CHECKCLOSE ( filenum                          ! i
                 ,[ tape-disposition ] );          ! i
```

Parameters

filenum input

INT:value

is the file number of an open file to be closed in the backup process.

tape-disposition input

INT:value

if present, specifies magnetic tape disposition, as follows:

tape-disposition.<13:15>

0 Rewind and unload, do not wait for completion

1 Rewind, take offline, do not wait for completion

2 Rewind, leave online, do not wait for completion

3 Rewind, leave online, wait for completion

4 Do not rewind, leave online

If omitted, 0 is used.

Condition Code Settings

These settings are obtained from the CLOSE procedure in the backup process; CHECKCLOSE establishes these settings in the primary process:

< (CCL) indicates that an invalid file number was supplied or that the backup process does not exist.

= (CCE) indicates that the CLOSE was successful.

> (CCG) does not return from CHECKCLOSE.

Considerations

- Identification of the backup process

  The system identifies the process to be affected by the CHECKCLOSE operation from the process's mom field in the process control block (PCB). For named process pairs, this field is automatically set up during the creation of a backup process.

- The condition code returned from CHECKCLOSE indicates the outcome of the CLOSE in the backup process.
- See Considerations on page 3-69.

CHECKDEALLOCATESEGMENT Procedure (Superseded by SEGMENT_DEALLOCATE_CHKPT_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The CHECKDEALLOCATESEGMENT procedure deallocates an extended data segment from use by the backup process in a process pair. It is called by the primary process when it is no longer needed by the backup process.

Syntax for C Programmers

This passive backup procedure is not supported in C programs. For a comparison of active backup and passive backup, see the *Guardian Programmer's Guide*.

Syntax for TAL Programmers

```
CALL CHECKDEALLOCATESEGMENT ( segment-id            ! i
                            ,[ flags ]              ! i
                            ,error );               ! o
```

Parameters

segment-id input

INT:value

is the segment number of the segment, as specified in the call to ALLOCATESEGMENT that created it.

flags input

INT:value

if present, has the form:

<0:14>
Must be 0.

<15> 1 Indicates that dirty pages in memory are not to be copied to the swap file (see ALLOCATESEGMENT Procedure (Superseded by SEGMENT_ALLOCATE_ Procedure) on page 2-20).

0 Indicates that dirty pages in memory are to be copied to the swap file.

If omitted, this parameter defaults to 0.

error output

INT .EXT:ref:1

indicates the outcome of the call. This procedure returns all values returned by DEALLOCATESEGMENT in the backup process, plus these file-system errors:

2 The segment ID is invalid or the backup process could not deallocate the segment.

29 The *segment-id* is missing.

30 No control blocks are available for linking.

31 Cannot use the process file segment (PFS), or the PFS has no room for a message buffer in either the backup process or the primary process.

201 Unable to link to the backup process.

Condition Code Settings

< (CCL) is set if the *error* parameter is missing, or a bounds error occurs on the *error* parameter.

= (CCE) is set by all other errors (see *error* parameter).

> (CCG) is never returned from this procedure.

Considerations

- Allocation by the primary process

  The segment need not be allocated by the primary process at the time of the call to CHECKDEALLOCATESEGMENT.

- *flags* parameter

  The *flags*.<15> = 1 option is used to improve performance when the swap file is a permanent file or a temporary file that is opened concurrently by an application. Following the call to CHECKDEALLOCATESEGMENT, the contents of the swap file are unpredictable. If the CHECKDEALLOCATESEGMENT call causes a purge of a temporary file, the system does not write the dirty pages (that is, pages that are being used) out to the file. If the *flags* parameter is missing, the default value of 0 is used.

- Segment deallocation

  When a segment is deallocated, the swap file end of file (EOF) is set to the larger of (1) the EOF when the file is opened by ALLOCATESEGMENT or (2) the end of the highest numbered page that is written to the swap file. All file extents beyond the EOF that did not exist when the file was opened are deallocated.

  Before deallocating a segment, this procedure removes all memory access breakpoints set in that segment.

- Shared segments

  A shared segment remains in existence until it has been deallocated by all the processes that allocated it.

CHECKDEFINE Procedure

Summary

This procedure is used to update a backup process with a DEFINE that was changed in the primary process.

Syntax for C Programmers

This passive backup procedure is not supported in C programs. For a comparison of active backup and passive backup, see the *Guardian Programmer's Guide*.

Syntax for TAL Programmers

```
status := CHECKDEFINE [ ( define-name ) ];     ! i
```

Parameters

status returned value

INT

returns a status word encoded as follows:

<0:7> = 0 Operation successful

<0:7> = 1 Could not communicate with backup process, then <8:15> = file-system error number

define-name input

STRING .EXT:ref:24

is the 24-byte array that contains the name of the DEFINE to be sent to the backup process. The name is left-justified and padded on the right with blanks. Trailing blanks are ignored.

Considerations

- If the *define-name* parameter is omitted, the working attribute set of the backup is updated to match that of the primary process.
- If the named DEFINE does not exist in the primary at the time of the call, then CHECKDEFINE will cause deletion of the DEFINE of the given name in the backup process if one exists. Otherwise, the named DEFINE will be copied to the backup, replacing the backup's version of the DEFINE if it has one.
- If the *define-name* parameter is supplied, but the first two bytes have the value 255 (-1 when treated as a word), then all DEFINEs in the backup process will be deleted.

- Note that since all DEFINEs are propagated to the backup process when it is created, use of CHECKDEFINE is not necessary unless one or more DEFINEs are changed.
- If a call to CHECKDEFINE causes a DEFINE in the backup to be altered, deleted, or added, then the context-change count for the backup process is incremented.

Example

```
STRING .EXT define^name[0:23];
LITERAL success = 0;
          .
          .
define^name ':=' ["=mydefine                  "];
status := CHECKDEFINE ( define^name );
IF status <> success THEN ... ;
```

CHECKMONITOR Procedure

Summary

The CHECKMONITOR procedure is called by a backup process to monitor the state of the primary process and to return control to the appropriate point (in the backup process) in the event of a failure of the primary process.

Syntax for C Programmers

This passive backup procedure is not supported in C programs. For a comparison of active backup and passive backup, see the *Guardian Programmer's Guide*.

Syntax for TAL Programmers

```
status := CHECKMONITOR;
```

Parameters

status returned value

INT

returns a status word of this form:

```
<0:7>  = 2

<8:15> = 0   Primary process stopped
         1   Primary process abnormally ended
         2   Primarys porcess processor failed
         3   Primary process called CHECKSWITCH
```

Note. The normal return from a call to CHECKMONITOR is to the statement following a call to the CHECKPOINT[MANY][X] procedure. The return corresponds to the latest call to CHECKPOINT[MANY][X] by the primary process in which its stack was checkpointed.

The backup process executes the statement following the call to CHECKMONITOR only if the primary process has not checkpointed its stack through a call to the CHECKPOINT[MANY][X] procedure.

Considerations

- Takeovers and selectable data segments in use

  If the stack has never been checkpointed, then at a takeover, the selectable segment in use at the time of the call to the CHECKMONITOR or CHECKSWITCH procedure is put into use. No segment is put into use if the segment is not available; that is, the SEGMENT_ALLOCATE_CHKPT_ or CHECKALLOCATESEGMENT procedure was not called to allocate the segment to the backup process, or the SEGMENT_DEALLOCATE_CHKPT_ or CHECKDEALLOCATESEGMENT procedure was called to deallocate the segment from the backup process.

Messages

If CHECKMONITOR (or another checkpointing procedure) returns a value indicating that a takeover has occurred due to a processor failure, the system subsequently delivers a system message -2 (processor down) to the $RECEIVE file of the new primary process. This might not be the first message delivered to the new primary process if other system messages have arrived since the last checkpoint operation of the old primary.

If CHECKMONITOR returns a value indicating that a takeover has occurred due to the primary process stopping, the new primary process receives a system message -101 (Process Deletion) on its $RECEIVE file.

For the format of system message -2 (processor down) or -101 (Process Deletion), see the *Guardian Procedure Errors and Messages Manual*.

CHECKOPEN Procedure (Superseded by FILE_OPEN_CHKPT_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The CHECKOPEN procedure is called by a primary process to open a designated file for its backup process. These two conditions must apply before the call to CHECKOPEN:

- The primary process must first open the file.
- The backup process must be in the “monitor” state (that is, in a call to CHECKMONITOR) for the CHECKOPEN to be successful.

The call to CHECKOPEN causes the CHECKMONITOR procedure in the backup process to call the file-system FILE_OPEN_ procedure for the designated file.

Syntax for C Programmers

This passive backup procedure is not supported in C programs. For a comparison of active backup and passive backup, see the *Guardian Programmer’s Guide*.

Syntax for TAL Programmers

```
CALL CHECKOPEN ( [ file-name ]                              ! i
                ,filenum                                    ! i
                ,[ flags ]                                  ! i
                ,[ sync-or-receive-depth ]                  ! i
                ,[ sequential-block-buffer-id ]             ! i
                ,[ buffer-length ]                          ! i
                ,backerror );                               ! o
```

Parameters

With the exception of *filenum*, all of the input parameters to this procedure are ignored; the values that were specified when the primary process called OPEN or FILE_OPEN_ are used instead. The ignored parameters are described under the OPEN procedure.

filenum output

INT:ref:1

is the number that identifies the file that was opened by the primary process and that is now to be opened by the backup process.

backerror output

INT:ref:1

returns one of these values:

Š0 is the file-system error number reflecting the call to FILE_OPEN_ in the backup process.

-1 indicates that the backup process is not running or that the checkpoint facility could not communicate with the backup process.

Condition Code Settings

These settings are obtained from the FILE_OPEN_ procedure in the backup process:

< (CCL) indicates that the open failed. The file-system error number returns in *backerror*.

= (CCE) indicates that the open was successful.

> (CCG) indicates that the open was successful, but an exceptional condition was detected. The file-system error number returns in *backerror*.

Considerations

- Identification of the backup process

  The system identifies the process to be affected by CHECKOPEN from the process's mom field in the process control block (PCB). For named process pairs, this field is automatically set up during the creation of a backup process.

- Nowait opens with CHECKOPEN

  If a process file is opened nowait (*flag*.<8> = 1 with OPEN, *options*.<1> = 1 with FILE_OPEN_), that file is CHECKOPEN nowait. CHECKOPEN returns errors detected in parameter specification and system data-space allocation in *backerror* and the operation is considered complete.

If no error is returned in *backerror*, the operation must be completed by a call to AWAITIO in the primary process. If you specify the *tag* parameter, the value returned by AWAITIO is -29D; the returned count and buffer address are undefined. If the condition code CCL is returned by AWAITIO, the file is automatically checkclosed by the checkpointing facility. For a nonprocess file or a process file that is opened in a waited manner, bit <8> of the *flag* parameter is reset internally to zero and ignored.

- Primary process open

  A *backerror* value of 17 is returned if a device or process being opened is not, in its own view, currently open by the primary process. This can occur, for example, after a device has been rought UP or DOWN.

- See “Considerations” for the OPEN procedure.
- Opening a Licensed object file with write or read-write access turns off the License attribute, even if opened by the Superid.

Messages

- Unable to communicate with backup

  If an “unable to communicate with backup” error occurs (that is, *backerror* = -1), it normally indicates either that the backup process does not exist or that a system resource problem exists.

CHECKPOINT Procedure (Superseded by CHECKPOINTX Procedure)

Summary

Note. This procedure cannot be called by native processes. Although this procedure is supported for TNS processes, it should not be used for new development.

The CHECKPOINT procedure is called by a primary process to send information about its current executing state to its backup process. The checkpoint information enables the backup process to recover from a failure of the primary process in an orderly manner. The backup process must be in the “monitor” state (that is, in a call to the CHECKMONITOR procedure) for the checkpoint to be successful.

Syntax for C Programmers

This passive backup procedure is not supported in C programs. For a comparison of active backup and passive backup, see the *Guardian Programmer's Guide*.

Syntax for TAL Programmers

```
status := CHECKPOINT

  ( [ stack-origin ], [ buffer-1 ], [ count-1 ]        ! i,i,i
                    , [ buffer-2 ], [ count-2 ]        ! i,i
                            .              .
                            .              .
                            .              .
                    , [ buffer-13 ], [ count-13 ] );   ! i,i
```

Parameters

status returned value

INT

returns a status word of this form:

<0:7> = 0 No error

<0:7> = 1 No backup process or unable to communicate with backup process; then <8:15> = file-system error number

<0:7> = 2 Takeover from primary process; then <8:15> =
0 Primary process stopped
1 Primary process abnormally ended
2 Primarys process processor failed
3 Primary process called CHECKSWITCH

<0:7> = 3 Invalid parameter; then <8:15> = number of parameter in error (leftmost position = 1)

stack-origin input

INT:ref:*

checkpoints the process's data stack from *stack-origin* through the current tip-of-stack location ('S'). A checkpoint of the data stack defines a restart point for the backup process.

buffer-n input

INT:ref:*

checkpoints a block of the process's data area (usually a file buffer) from *buffer-n* for the number of words specified by the corresponding *count-n* parameter. If

you omit *buffer-n*, *count-n* is treated as *filenum* and that file's file synchronization block is checkpointed.

count-n input

INT:value

The use of this parameter depends on the presence or absence of the corresponding *buffer-n* parameter:

- If *buffer-n* is present, then *count-n* specifies the number of words to be checkpointed.
- If *buffer-n* is absent, then *count-n* is the *filenum* of a file whose file synchronization block is to be checkpointed.

Note. If the message is too large (that is, the total of the stack size and the counts of all buffers and the size of all file synchronization blocks is too big), *status*.<0:7> is set to 3 and the parameter number is set to 26.

Considerations

- Checkpointing the process's data stack

  The CHECKPOINT procedure provides for checkpointing the process's data stack and any combination of up to 13 separate data blocks and file synchronization blocks. A data block can be from any location in the data area. (Data blocks are usually file buffers that are not checkpointed as part of the stack, and they cannot be in an extended data area.)

- Maximum checkpoint size

  The largest stack area or data item that can be checkpointed is 32,500 bytes. Additionally, the sum total of the sizes of the stack area and each checkpoint item, plus an allowance of 20 bytes for each item, should not exceed 32,500 bytes. An item in this context means either a data item (user-declared size) or a file synchronization block with varying sizes.

- Identification of the backup process

  The system identifies the process to be affected by the CHECKPOINT operation from the process's mom field in the process control block (PCB). For named process pairs, this field is automatically set up during the creation of a backup process.

- Checkpointing a file's synchronization (sync) block

  If a file's sync block is checkpointed, the call to CHECKPOINT contains an implicit call to the GETSYNCINFO procedure for the file. Therefore, checkpointing of a file's sync block should not be performed between an I/O completion and a call to the FILE_GETINFO_ (or FILEINFO) procedure for that file. If file sync block

checkpointing is performed, FILE_GETINFO_ returns (in its *last-error* parameter) the status of the call to GETSYNCINFO (usually, *last-error* = 0).

- Unable to communicate with backup

  If an “unable to communicate with backup” error (that is, *status*.<0:7> = 1) occurs, this normally indicates either that the backup process does not exist or that a system resource problem exists. If a system resource problem exists, the checkpoint message to the backup is probably too large.

- Invalid parameter

  If you attempt to checkpoint the data area in the region used by the CHECKMONITOR procedure in the backup process, then CHECKPOINT returns an “invalid parameter” error (that is, *status*.<0:7> = 3). See the recovery procedure in Considerations on page 3-33.

- Takeovers and selectable segments

  The selectable segment put into use following takeover depends on several factors:

  - The segment in use at the time of the last checkpoint is put into use if it is available; that is, the segment was allocated to the backup using the SEGMENT_ALLOCATE_CHKPT_ or CHECKALLOCATESEGMENT procedure and has not since been deallocated by the SEGMENT_DEALLOCATE_CHKPT_ or CHECKDEALLOCATESEGMENT procedure.
  - The segment in use when the CHECKMONITOR or CHECKSWITCH procedure was called is used if the segment in use at the time of the last checkpoint is no longer available.
  - No segment is used if the segment in use at the time of the last checkpoint and the segment in use when the CHECKMONITOR or CHECKSWITCH procedure was called are both unavailable.

CHECKPOINTMANY Procedure (Superseded by CHECKPOINTMANYX Procedure)

Summary

Note. This procedure cannot be called by native processes. Although this procedure is supported for TNS processes, it should not be used for new development.

The CHECKPOINTMANY procedure (like the CHECKPOINT procedure) is called by a primary process to send information about its current executing state to its backup process.

The CHECKPOINTMANY procedure is used in place of CHECKPOINT when there are more than 13 pieces of information to be sent.

Syntax for C Programmers

This passive backup procedure is not supported in C programs. For a comparison of active backup and passive backup, see the *Guardian Programmer's Guide*.

Syntax for TAL Programmers

```
status := CHECKPOINTMANY ( [ stack-origin ]          ! i
                         ,[ descriptors ] );     ! i
```

Parameters

status returned value

INT

returns a status word of this form:

<0:7> = 0 No error

<0:7> = 1 No backup process or unable to communicate with backup process; then <8:15> = file-system error number.

<0:7> = 2 Takeover from primary; then <8:15> =

0	Primary process stopped
1	Primary process abnormally ended
2	Primarys process processor failed
3	Primary process called CHECKSWITCH

<0:7> = 3 Invalid parameter, then <8:15> =

1	Error in *stack-origin* parameter
n > 1	Error in word [n-2] (see “Considerations”)

stack-origin input

INT:ref:*

contains an address. CHECKPOINTMANY checkpoints the process’s data stack from *stack-origin* through the current tip-of-stack location (‘S’). A checkpoint of the data stack defines a restart point for the backup process.

descriptors input

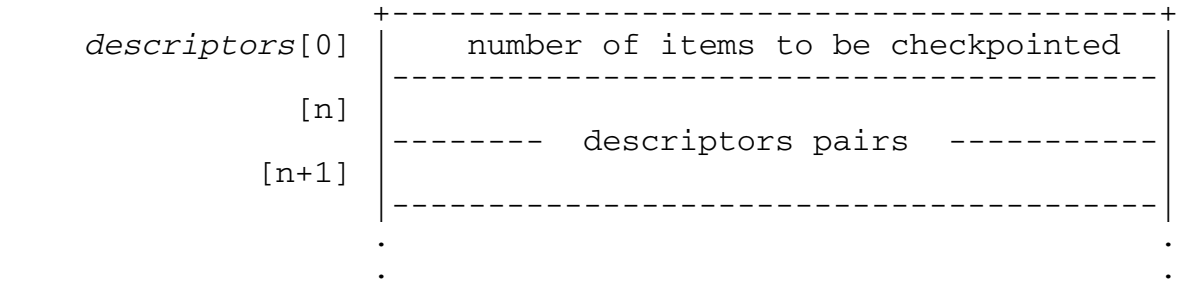
INT:ref:*

is an array that describes the items (data blocks or file synchronization blocks) to be checkpointed. The first word of the array, *descriptors*[0], is a count of the number of items to be checkpointed. The rest of the array consists of pairs of words, each pair describing one of the items. (See “Considerations.”)

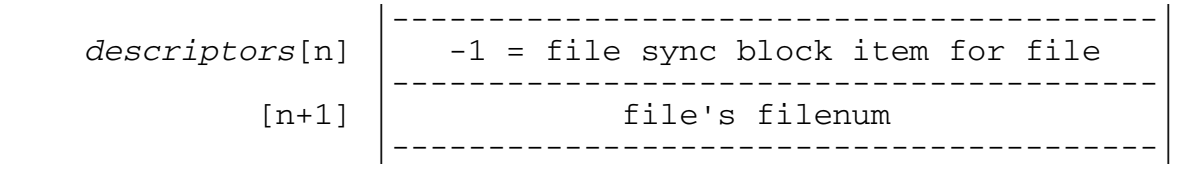
Considerations

- *descriptors* array form:

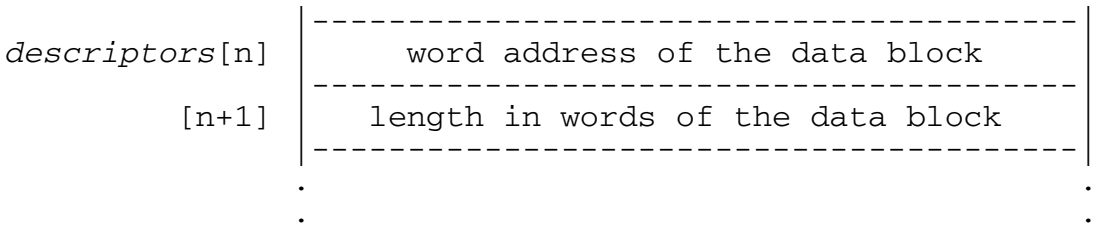
  Following word zero, *descriptors* consists of pairs of words.



  If the first word of the pair contains -1, the pair describes a file synchronization block item for the file whose file number is in the second word of the pair.



Otherwise, the pair of words describes a data block to be checkpointed: the first word contains the word address of the data block, and the second word contains the length, in words, of the data block:



The size, in words, of the *descriptors* array must be at least

```
1 + 2 * descriptors[0]
```

- Invalid parameter location

  If *status*.<0:7> = 3, then *status*.<8:15> has this meaning:

  status.<8:15> = 1error in *stack-origin* parameter
  = n, n > 1error in word [n-2]

  If the *descriptors* pair describes a file synchronization block (first word of pair = -1, second word of pair = file number), then:

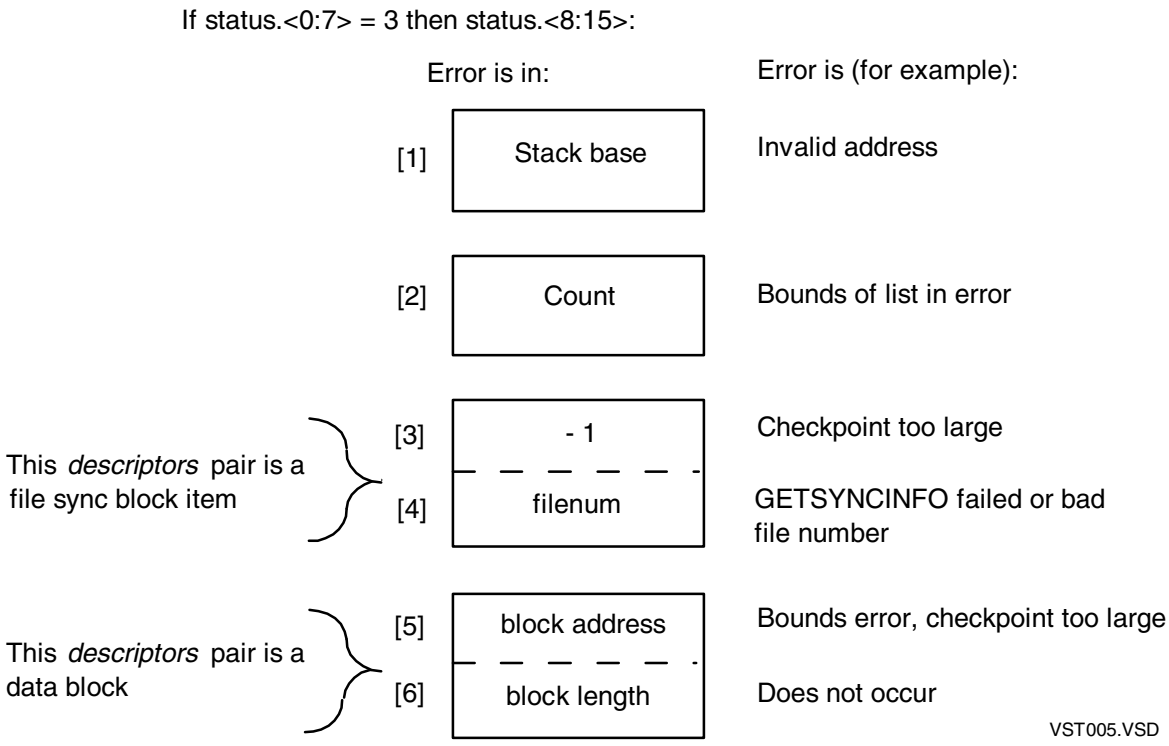
  - If the filesync block makes the checkpoint exceed 32,500 bytes, then descriptors [n-2] is the first word of the pair.
  - If any other error occurs (such as GETSYNCINFO fails or bad file number), then descriptors [n-2] is the second word of the pair.

  If the pair describes a buffer (first word = address, second word = length), then:

  - If the address, or the address plus the length, results in a bounds violation, then *descriptors*[n-2] is the first word of the pair.
  - If this buffer makes the total amount of data to be checkpointed (data + sync blocks + stack) exceed 32,500 bytes, then descriptors [n-2] is the first word of the pair.

For example:

Figure 3-1. Invalid Parameter Location



- Maximum checkpoint size

  The largest stack area or data item that can be checkpointed is 32,500 bytes. Additionally, the sum total of the sizes of the stack area and each checkpoint item, plus an allowance of 20 bytes for each item, should not exceed 32,500 bytes. An item in this context means either a data item (of user declared size) or a file synchronization block of varying sizes.

- The CHECKPOINTMANY procedure allows checkpointing of both the process's data stack and any number of blocks.

- Identification of the backup process

  The system identifies the process to be affected by the CHECKPOINTMANY operation from the process's mom field in the process control block (PCB). For named process pairs, this field is automatically set up during the creation of a backup process.

- Invalid parameter

  If an attempt is made to checkpoint the data area used by CHECKPOINTMANY for system-oriented stack maintenance, it returns an "invalid parameter" error (that is, *status*.<0:7> = 3).

- Takeovers and selectable segments

  The selectable segment put into use following takeover depends on several factors:

  - The segment in use at the time of the last checkpoint is put into use if it is available; that is, the segment was allocated to the backup using the SEGMENT_ALLOCATE_CHKPT_ or CHECKALLOCATESEGMENT procedure and has not since been deallocated by the SEGMENT_DEALLOCATE_CHKPT_ or CHECKDEALLOCATESEGMENT procedure.
  - The segment in use when the CHECKMONITOR or CHECKSWITCH procedure was called is used if the segment in use at the time of the last checkpoint is no longer available.
  - No segment is used if the segment in use at the time of the last checkpoint and the segment in use when the CHECKMONITOR or CHECKSWITCH procedure was called are both unavailable.

- See also Considerations on page 3-38 for the CHECKPOINT procedure.

Example

```
DESCRIPTORS[0] := 2;           ! count of items.
DESCRIPTORS[1] := -1;          ! sync item.
DESCRIPTORS[2] := FNUM^A;      ! file number.
DESCRIPTORS[3] := @BUFFER;     ! data item: word address.
DESCRIPTORS[4] := 512;         ! number of words.
 STAT:= CHECKPOINTMANY( STK^ORIGIN , DESCRIPTOR);
    !  this is equivalent to:
    !  STAT := CHECKPOINT( STK^ORIGIN, , FNUM^A, BUFFER, 512 );
```

CHECKPOINTMANYX Procedure

Summary

The CHECKPOINTMANYX procedure (like the CHECKPOINTX procedure) is called by a primary process to send information about its current executing state to its backup process. The checkpoint information enables the backup process to recover from a failure of the primary process in an orderly way. The backup process must be in the “monitor” state (that is, in a call to the CHECKMONITOR procedure) for the CHECKPOINTMANYX call to be successful.

This procedure can be used to checkpoint:

- Stack data from a specified stack marker to the tip of the stack
- Multiple data areas
- File synchronization blocks

The CHECKPOINTMANYX procedure can be used by both TNS processes and native processes. It allows checkpointing of data in extended data segments (flat or selectable) in addition to the user data segment.

You must use CHECKPOINTMANYX if you need to checkpoint more than five data areas. You can use the CHECKPOINTX procedure instead if you need to checkpoint five or fewer data areas.

Syntax for C Programmers

This passive backup procedure is not supported in C programs. For a comparison of active backup and passive backup, see the *Guardian Programmer's Guide*.

Syntax for TAL Programmers

```
status := CHECKPOINTMANYX ( [ stack-origin ]        ! i
                          ,[ descriptors ] );       ! i
```

Parameters

status returned value

INT

returns a status word of this form:

<0:7> = 0 No error

<0:7> = 1 No backup process or unable to communicate with backup process; then <8:15> = file-system error number

<0:7> = 2 Takeover from primary process; then <8:15> =

0 Primary process stopped
1 Primary process abnormally ended
2 Primarys process processor failed
3 Primary process called CHECKSWITCH

<0:7> = 3 Invalid parameter; then <8:15> =

1 Error in *stack-origin* parameter
2 Bounds error on *descriptors*
>2 Error in specified descriptor. If bits <8:15> = 3 then the first descriptor is in error. If <8:15> = 4 then the second descriptor is in error, and so on.

stack-origin input

INT:ref:*

contains an address. CHECKPOINTMANYX checkpoints the process's data stack from the address in *stack-origin* through the current tip-of-stack location ('S'). A checkpoint of the data stack defines a restart point for the backup process.

See "Considerations" for details.

descriptors input

INT .EXT:ref:*

is an array that describes the items (data blocks or file synchronization blocks) to be checkpointed. The first word of the array, *descriptors*[0], contains the number of items to be checkpointed. The rest of the array consists of sets of words, with each set containing five words and describing one item. (See "Considerations.")

Its format is as follows:

[0]	Contains the number of items to be checkpointed. Each item consists of a set of five words.

If the item is a file:

[0:1]	Equals -1D.
[2]	Is the file number.
[3]	(Is reserved.)
[4]	(Is reserved.)

If the item is a data area:

[0:1]	Is the length in bytes.
[2]	Is the segment ID. (If it is -1, the address is in the stack or current segment.)
[3:4]	Is the extended address.

Considerations

- Checkpointing the stack

  Checkpointing the entire data stack has the effect of providing a restart point for the backup process. The *stack-origin* parameter gives you the option of specifying how far into the stack to start checkpointing. Although native stacks grow downward while TNS stacks grow upward, the effect is the same—all data from *stack-origin* to the growing tip of the stack is checkpointed.

The rules for specifying the *stack-origin* address, however, are different for TNS processes and native processes. In a TNS process, you can include global variables to be checkpointed with the stack data, because the global variables immediately precede the stack; thus you can checkpoint all global variables with the stack by specifying a *stack-origin* address of zero (0).

In a native process, you cannot checkpoint global data with the stack, because global variables are not adjacent to the stack. If the *stack-origin* parameter is specified for a native process, it must point to a location within the data stack itself. To checkpoint global data, you must do so explicitly by providing a descriptor for each area of global data you want to checkpoint. Note that this approach works for TNS processes as well as for native processes; therefore a program written this way can be compiled for either architecture.

Establishing the *stack-origin* address can be done in several ways. These approaches work for both TNS and native processes:

- To checkpoint the entire stack, set the *stack-origin* address to -3. In a TNS process this value is the equivalent of the initial -L value in the TNS stack area. In a native process this value indicates the start of the main stack.
- To checkpoint starting from the origin of an arbitrary procedure, introduce a lower procedure to obtain its stack address. For example, assume a procedure MYPROC is to be the base procedure for a stack checkpoint; you can obtain its stack address in a global pointer STACKBASE as follows:

```
INT  .STACKBASE;

PROC SET_MYPROC_BASE;
BEGIN
   INT  .DUMMY;
   @STACKBASE := @DUMMY;
   CALL MYPROC;
END;
```

  The *stack-origin* address (if you do not specify the value -3) designates the boundary between what is to be checkpointed with the stack and what is not. In native processes, which use descending stacks, the address is that of the first byte not to be checkpointed. In a TNS process, it is the address of the first byte included in the checkpointed data.

Other methods of establishing the *stack-origin* address work only with TNS processes. These methods include:

- Refer to the L register.
- Pick up the address of a local variable other than as described above for a procedure call. This approach does not work for native processes because the location of local variables in the enclosing stack frame is not defined by the compiler other than by inclusion.

If the *stack-origin* parameter is omitted, the stack is not checkpointed. You can, however, include the *stack-origin* parameter without checkpointing the stack by setting *stack-origin* to -1.

- Checkpointing data areas

  Checkpointing specific variables involves using the *descriptors* parameter to specify the addresses of data areas and the number of bytes to be checkpointed. Differences in data layout between a TNS stack and a native main stack cause some restrictions in the way native processes address these buffers. These rules apply to native processes. Code that follows these rules can be compiled to run as either a TNS process or as a native process:

  - To checkpoint global variables, refer to the variables themselves. Do not use constant addresses.
  - If your program depends on two global variables being adjacent, you must ensure that they are in a data block together. In pTAL, this is done automatically if blocks are not explicit and if the BLOCKGLOBALS compiler directive is not used.
  - Do not assume adjacency or order of local variables; use structures or arrays
  - Use $LEN or an equivalent language function to determine the length of data items. The lengths of some data items differ between a TNS process and a native process.

  When checkpointing a set of global variables, if the set is small enough, you can obtain their address and size using the PROCESS_GETINFOLIST_ procedure, items 108 and 109.

  Code that will only be run as a TNS process can use constants for addressing global variables and assume adjacency of variables.

- Do not checkpoint heap or pool storage

  Native processes can use a standard heap area for dynamic memory allocation; programs using the Common Run-Time Environment (CRE) make this heap available, for example, by using the C `malloc()` function. A TNS process can achieve a similar effect with a flat segment that has space structured as a standard memory pool.

  Process pairs should not checkpoint data residing in the heap or memory pool. Control information is needed to maintain structure, and this control information can be neither obtained nor checkpointed. If the backup process were to take over using checkpointed heap or pool data, its heap or pool would be corrupt and allocations and deallocations would not work. Not only would the space control information be corrupt, but the backup typically would not even have underlying memory allocated at the needed address to receive the data at the time of the checkpoint.

- Checkpoint message size limit

The largest stack area or data item that can be checkpointed is 32,500 bytes. Additionally, the sum total of the sizes of the stack area and each checkpoint item, plus an allowance of 20 bytes for each item, should not exceed 32,500 bytes. An item in this context means either a data item (user-declared size) or a file synchronization block with varying sizes.

For native processes, the size of the checkpoint message sent to the backup process is limited to 50,000 bytes. This additional message capacity is necessary because of increased data memory requirements.

The extra space in a checkpoint message for a native process enables TNS process pairs to be converted to native processes and allows a program to be compiled for either environment.

- If the address is in an extended data segment, the backup must also have that extended data segment allocated. The backup must have the same segment ID, and the segment should be the same size. If the backup has a smaller size, any data in the primary that is outside of the addressable area of the segment in the backup is not checkpointed. If the backup does not have a segment with that segment ID, an error is returned and no data or file information is checkpointed.
- Extended addresses must be relative; they cannot be absolute. Extended addresses cannot be in the user code space.
- Takeovers and selectable segments

  The selectable segment put into use following takeover depends on several factors:

  - The segment in use at the time of the last checkpoint is put into use if it is available; that is, the segment was allocated to the backup process using the SEGMENT_ALLOCATE_CHKPT_ or CHECKALLOCATESEGMENT procedure and has not since been deallocated by the SEGMENT_DEALLOCATE_CHKPT_ or CHECKDEALLOCATESEGMENT procedure.
  - The segment in use when the CHECKMONITOR or CHECKSWITCH procedure was called is used if the segment in use at the time of the last checkpoint is no longer available.
  - No segment is used if the segment in use at the time of the last checkpoint and the segment in use when the CHECKMONITOR or CHECKSWITCH procedure was called are both unavailable.
- Do not try to checkpoint data in a read-only segment.
- You can checkpoint data in shared extended data segments, but you must ensure consistency of the data among all processes that might be sharing the segment, both in the primary processor and the backup processor.
- Stack allocation for native processes

The backup process can abnormally terminate if not enough disk or memory resources are available to increase the size of the main stack in the backup process. This situation is possible in a native process, because the main stack is allocated dynamically or on request. By contrast, TNS stacks are statically allocated.

Use the space guarantee attribute of the object file or process creation procedure (PROCESS_LAUNCH_ or PROCESS_SPAWN_) to ensure that enough resources are available when the native process is created.

- Errors returned by CHECKPOINTMANYX

  CHECKPOINTMANYX returns these errors:

  - The checkpoint message contains a buffer in an extended data segment, and the backup does not have that segment allocated (file-system error 22).
  - The backup process does not exist.
  - Parameter errors (*status*.<0:7> = 3):
    - A bounds error occurred on the *descriptor* array.
    - The file number is not open.
    - The extended address is absolute.
    - The extended address is in logical segment 1, 2, or 3 (code or library spaces); that is, not in a data segment or the stack.
    - The segment ID was equal to -1 and the address was in an extended data segment, but no selectable segment was in use at the time of the call to CHECKPOINTMANYX.
    - The address was in the stack, but either the count was too large, the area was above the highest stack address, the area was beyond the end of the stack, or the area overlapped the area used by the CHECKMONITOR procedure.
    - The address was invalid; for example, the address was in an extended data segment, but either the segment ID was not allocated, the segment ID was an invalid segment number, or there was a bounds error on the area.
    - The total message size was too large (over 32 KB).

Example

```
INT status;
INT stack^origin;
INT junk;
STRING .EXT buffer[0:511];
INT .EXT descr[0:10];

descr[0]    := 2;              ! count of items
```

```
! note the following is improper syntax;
! used for illustration only

descr[1:2]  := -1D;           ! always this for file items
descr[3]    := fnum^a;        ! file number
descr[4:5]  := junk;          ! unused words for file items

descr[6:7]  := 512D;          ! length in bytes
descr[8]    := -1;            ! indicates stack
descr[9:10] := @buffer;       ! data item -- extended address

status := CHECKPOINTMANYX( stk^origin , descr);
```

CHECKPOINTX Procedure

Summary

The CHECKPOINTX procedure (like the CHECKPOINTMANYX procedure) is called by a primary process to send information about its current executing state to its backup process. The checkpoint information enables the backup process to recover from a failure of the primary process in an orderly manner. The backup process must be in the “monitor” state (that is, in a call to the CHECKMONITOR procedure) for the CHECKPOINTX call to be successful.

This procedure can be used to checkpoint:

- Stack data from a specified stack address to the tip of the stack
- Up to five data areas
- File synchronization blocks

The CHECKPOINTX procedure can be used by both TNS processes and native processes. It allows checkpointing of data in extended data segments (flat or selectable) in addition to the user data segment.

Use the CHECKPOINTMANYX procedure if you need to checkpoint more than five data areas.

Syntax for C Programmers

This passive backup procedure is not supported in C programs. For a comparison of active backup and passive backup, see the *Guardian Programmer's Guide*.

Syntax for TAL Programmers

```
status := CHECKPOINTX ( [ stack-origin ]                    ! i

    ,[ segment-id1 ], [ bufferx-1 ], [ count-1 ]            ! i,i,i
    ,[ segment-id2 ], [ bufferx-2 ], [ count-2 ]            ! i,i,i
                              .              .
                              .              .
                              .              .
    ,[ segment-id5 ], [ bufferx-5 ], [ count-5 ] );         ! i,i,i
```

Parameters

status returned value

INT

returns a status word of this form:

<0:7> = 0 No error

<0:7> = 1 No backup or unable to communicate with backup; then <8:15> = file-system error number

<0:7> = 2 Takeover from primary process; then <8:15> =

0 Primary process stopped
1 Primary process abnormally ended
2 Primarys process processor failed
3 Primary process called CHECKSWITCH

<0:7> = 3 Invalid parameter; then <8:15> = number of parameter in error:

1 *stack-origin* parameter
2 Parameter set 1
3 Parameter set 2
4 Parameter set 3
5 Parameter set 4

6 Parameter set 5
7 Total message too large

stack-origin input

INT:ref:*

contains an address. CHECKPOINTX checkpoints the process's data stack from the address in *stack-origin* through the current tip-of-stack location ('S'). A checkpoint of the data stack defines a restart point for the backup process.

See "Considerations" for details.

segment-idn input

INT:value

contains the segment ID of the extended data segment if the *bufferx-n* parameter is provided and the data block to be checkpointed is in an extended data segment. If *segment-idn* is omitted or equal to -1, the data block is assumed to be either in the flat segment, in the selectable segment currently in use, or on the stack, depending on the address provided.

If *bufferx-n* is omitted, *segment-id* contains the file number of a file whose file synchronization block is to be checkpointed. The *count-n* parameter is ignored in this case.

bufferx-n input

STRING .EXT:ref:*

is the address of the data area to be checkpointed. See "Considerations" for details.

If *bufferx-n* is omitted, a file synchronization block is to be checkpointed and the file number is specified in the *segment-idn* parameter.

count-n input

INT(32):value

contains the number of bytes to be checkpointed if *bufferx-n* is provided.

If *bufferx-n* is omitted, this parameter is ignored.

Considerations

- Checkpointing the stack

  Checkpointing the entire data stack has the effect of providing a restart point for the backup process. The *stack-origin* parameter gives you the option of specifying how far into the stack to start checkpointing. Although native stacks grow downward while TNS stacks grow upward, the effect is the same—all data from *stack-origin* to the tip of the stack is checkpointed.

The rules for specifying the *stack-origin* address, however, are different for TNS processes and native processes. In a TNS process, you can include global variables to be checkpointed with the stack data, because the global variables immediately precede the stack; thus you can checkpoint all global variables with the stack by specifying a *stack-origin* address of zero (0).

In a native process, you cannot checkpoint global data with the stack, because global variables are not adjacent to the stack. If the *stack-origin* parameter is specified for a native process, it must point to a location within the data stack itself. To checkpoint global data, you must do so explicitly using the *bufferx-n* and *count-n* parameters. Note that this approach works for TNS processes as well as for native processes; therefore a program written this way can be compiled for either architecture.

Establishing the *stack-origin* address can be done in several ways. These approaches work for both TNS and native processes:

- To checkpoint the entire stack, set the *stack-origin* address to -3. In a TNS process this value is the equivalent of the initial -L value in the TNS stack area. In a native process this value indicates the start of the main stack.
- To checkpoint starting from the origin of an arbitrary procedure, introduce a lower procedure to obtain its stack address. For example, assume a procedure MYPROC is to be the base procedure for a stack checkpoint; you can obtain its stack address in a global pointer STACKBASE as follows:

```
INT  .STACKBASE;

PROC SET_MYPROC_BASE;
BEGIN
   INT  .DUMMY;
   @STACKBASE := @DUMMY;
   CALL MYPROC;
END;
```

  The *stack-origin* address (if you do not specify the value -3) designates the boundary between what is to be checkpointed with the stack and what is not. In native processes, which use descending stacks, the address is that of the first byte not to be checkpointed. In a TNS process, it is the address of the first byte included in the checkpointed data.

Other methods of establishing the *stack-origin* address work only with TNS processes. These methods include:

- Refer to the L register.
- Pick up the address of a local variable other than as described above for a procedure call. This approach does not work for native processes because the location of local variables in the enclosing stack frame is not defined by the compiler.

If the *stack-origin* parameter is omitted, the stack is not checkpointed. You can, however, include the *stack-origin* parameter without checkpointing the stack by setting *stack-origin* to -1.

- Checkpointing data areas

  Checkpointing specific variables involves specifying the address of a data area in the *bufferx-n* parameter and a byte count in *count-n*. Differences in data layout between a TNS stack and a native main stack cause some restrictions in the way native processes address these buffers. These rules apply to native processes. Code that follows these rules can be compiled to run as either a TNS process or a native process:

  - To checkpoint global variables, refer to the variables themselves. Do not use constant addresses.
  - If your program depends on two global variables being adjacent, you must ensure that they are in a data block together. In pTAL, this is done automatically if blocks are not explicit and if the BLOCKGLOBALS compiler directive is not used.
  - Do not assume adjacency or order of local variables; use structures or arrays.
  - Use $LEN or an equivalent language function to determine the length of data items, and use this value in the *count-n* parameter. The lengths of some data items differ between a native process and a TNS process.

  When checkpointing a set of global variables, if the set is small enough, you can obtain their address and size using the PROCESS_GETINFOLIST_ procedure, items 108 and 109.

  Code that will only be run as a TNS process can use constants for addressing global variables and assume adjacency of global and local variables.

- Do not checkpoint heap or pool storage

  Native processes can use a standard heap area for dynamic memory allocation; programs using the Common Run-Time Environment (CRE) make this heap available, for example, by using the C `malloc()` function. A TNS process can achieve a similar effect with a flat segment that has space structured as a standard memory pool.

  Process pairs should not checkpoint data residing in the heap or memory pool. Control information is needed to maintain structure, and this control information can be neither obtained nor checkpointed. If the backup process were to take over using checkpointed heap or pool data, its heap or pool would be corrupt and allocations and deallocations would not work. Not only would the space control information be corrupt, but the backup typically would not even have underlying memory allocated at the needed address to receive the data at the time of the checkpoint.

- Checkpoint message size limit

  The largest stack area or data item that can be checkpointed is 32,500 bytes. Additionally, the sum total of the sizes of the stack area and each checkpoint item, plus an allowance of 20 bytes for each item, should not exceed 32,500 bytes. An item in this context means either a data item (user-declared size) or a file synchronization block with varying sizes.

  For native processes, the size of the checkpoint message sent to the backup process is limited to 50,000 bytes. This additional message capacity is necessary because of increased data memory requirements.

  The extra space in a checkpoint message for a native process enables TNS process pairs to be converted to native processes and allows a program to be compiled for either environment.

- If the address is in an extended data segment, the backup must also have that extended data segment allocated. The backup must have the same segment ID, and the segment should be the same size. If the backup has a smaller size, any data in the primary process that is outside of the addressable area of the segment in the backup process is not checkpointed. If the backup process does not have a segment with that segment ID, an error is returned and no data or file information is checkpointed.

- Extended addresses must be relative; they cannot be absolute. Extended addresses cannot be in the user code space.

- Takeovers and selectable segments

  The selectable segment put into use following takeover depends on several factors:

  - The segment in use at the time of the last checkpoint is put into use if it is available; that is, the segment was allocated to the backup using the SEGMENT_ALLOCATE_CHKPT_ or CHECKALLOCATESEGMENT procedure and has not since been deallocated by the SEGMENT_DEALLOCATE_CHKPT_ or CHECKDEALLOCATESEGMENT procedure.

  - The segment in use when the CHECKMONITOR or CHECKSWITCH procedure was called is used if the segment in use at the time of the last checkpoint is no longer available.

  - No segment is used if the segment in use at the time of the last checkpoint and the segment in use when the CHECKMONITOR or CHECKSWITCH procedure was called are both unavailable.

- The CHECKPOINT procedure allows 13 items to be checkpointed at once and uses word counts, while CHECKPOINTX allows 5 items to be checkpointed at once and uses byte counts. The CHECKPOINT procedure cannot be called from a native process.

- Do not try to checkpoint data in a read-only segment.
- You can checkpoint data in shared extended data segments, but you must ensure consistency of the data among all processes that might be sharing the segment, both in the primary processor and in the backup processor.
- Stack allocation for native processes

  The backup process can abnormally terminate if not enough disk or memory resources are available to increase the size of the main stack in the backup process. This situation is possible in a native process, because the main stack is allocated dynamically or on request. By contrast, TNS stacks are statically allocated.

  Use the SPACE_GUARANTEE attribute of the object file or process creation procedure to ensure that enough resources are available when the native process is created.
- Errors returned by CHECKPOINTX

  CHECKPOINTX returns these errors:

  - The checkpoint message contains a buffer in an extended data segment, and the backup does not have that segment allocated (file-system error 22).
  - The backup process does not exist.
  - Parameter errors (*status*.<0:7> = 3):
    - The file number is not open.
    - The extended address is absolute.
    - The extended address is in logical segment 1, 2, or 3 (code or library spaces); that is, not in a data segment or the stack.
    - The segment ID was omitted or was equal to -1 and the address was in a selectable extended data segment, but no selectable segment was in use at the time of the call to CHECKPOINTX.
    - The address was in the stack, but either *count-n* was too large, the area was above the highest stack address, the area was beyond the end of stack, or the area overlapped with the area used by the CHECKMONITOR procedure.
    - The address was in an extended data segment, but either the segment ID was not allocated, the segment ID was an invalid segment number, or there was a bounds error on the area.
    - The total message size was too large (over 32.5 kilobytes for a TNS process or 50 kilobytes for a native process).

- An invalid combination of parameters occurred:

  There was a count, but no buffer; or
  there was a buffer, but no count; or
  there was a buffer and a segment ID, but no count.

CHECKRESIZESEGMENT Procedure

Summary

This procedure complements the RESIZESEGMENT procedure.

Syntax for C Programmers

This passive backup procedure is not supported in C programs. For a comparison of active backup and passive backup, see the *Guardian Programmer's Guide*.

Syntax for TAL Programmers

```
CALL CHECKRESIZESEGMENT ( segment-id,          ! i
                          error );             ! o
```

Parameters

segment-id input

INT:value

is the identifier for the extended data segment to be resized in the backup process. The size is taken from the current size of *segment-id* in the primary process and *segment-id* must have been previously allocated in the primary process and the backup process.

error output

INT .EXT:ref:1

returns a file-system error code indicating the outcome of the call, one of:

2	Segment not allocated by the primary process or segment ID is invalid.
29	The *segment-id* is missing.
30	No control blocks available for linking.

`31` Cannot use the process file segment (PFS), or the PFS has no room for a message buffer in either the backup process or the primary process.

`201` Unable to link to the backup.

Other errors are returned from RESIZESEGMENT in the backup.

Condition Code Settings

< (CCL) is returned if the *error* parameter is missing or there is a bounds error on the *error* parameter.

= (CCE) indicates any condition not set by CCL.

> (CCG) is not returned from this procedure.

CHECKSETMODE Procedure

Summary

CHECKSETMODE allows a primary process of a process pair to propagate SETMODE operations to the backup process of the pair.

Syntax for C Programmers

This passive backup procedure is not supported in C programs. For a comparison of active backup and passive backup, see the *Guardian Programmer's Guide*.

Syntax for TAL Programmers

```
CALL CHECKSETMODE ( filenum          ! i
                  ,function          ! i
                  ,error );          ! o
```

Parameters

filenum input

INT:value

is the number of an open file that identifies the file to receive the SETMODE *function*.

function input

INT:value

is one of these SETMODE functions:

12	Set terminal access mode. (The value specified in *parm2*.<15> of the primary process's SETMODE request is passed to the backup process.)
30	Allow nowait I/O operations to complete in any order.
36	Allow requests to be queued on $RECEIVE based on process priority.
71	Set transmission priority.
72	Force system buffering for nowait files.
80	Set system message modes.
117	Set TRANSID forwarding.
141	Enable/disable large transfers.
149	Set alternate key insertion locking.

error output

INT .EXT:ref:1

the error that occurred on the operation. These file-system errors are returned from CHECKSETMODE:

2	The *function* parameter is not one of the allowed values.
29	The *filenum* or *function* parameter is missing.
30	No message control blocks are available.
31	Cannot use the process file segment (PFS), or the PFS has no room for a message buffer in either the backup process or the primary process.
201	Unable to link to the backup process.

Condition Code Settings

If the *error* parameter is missing, or there is a bounds error on the *error* parameter, the condition code is set to CCL. All other errors set the condition code to CCE. CCG is never returned from this procedure.

Considerations

- CHECKSETMODE supports SETMODE functions that set flags in either the ACB of a file or the PCB of a process. The values of the flags set in the primary process's ACB or PCB set the backup process's flags.

- The caller of CHECKSETMODE is suspended until the operation is complete (even if the file is opened in nowait mode).

CHECKSWITCH Procedure

Summary

The CHECKSWITCH procedure is called by a primary process to cause the duties of the primary and backup processes to be interchanged.

The call to CHECKSWITCH contains an implicit call to the CHECKMONITOR procedure, so that the caller becomes the backup and monitors the execution state of the new primary process. The backup process must be in the monitor state (that is, in a call to CHECKMONITOR) for the CHECKSWITCH call to be successful.

Syntax for C Programmers

This passive backup procedure is not supported in C programs. For a comparison of active backup and passive backup, see the *Guardian Programmer's Guide*.

Syntax for TAL Programmers

```
status := CHECKSWITCH;
```

Parameters

status returned value

INT

returns a status word of this form:

<0:7> = 1 Could not communicate with backup process, then <8:15> = file-system error number

<0:7> = 2 then <8:15> =

0 Primary process stopped
1 Primary process abnormally ended

2 Primarys process processor failed
3 Primary process called CHECKSWITCH

Note. The normal return from a call to CHECKSWITCH is to the statement following a call to the CHECKPOINT[MANY][X] procedure. The return corresponds to the latest call to CHECKPOINT[MANY][X] by the primary process in which its stack was checkpointed.

The backup process executes the statement following the call to CHECKSWITCH only if the primary process has not checkpointed its stack through a call to CHECKPOINT[MANY][X].

Considerations

- When to use CHECKSWITCH

  Use CHECKSWITCH following the reload of a processor module. The purpose is to switch the process pair's work back to the original primary processor module. CHECKSWITCH causes the current backup to become the primary process and to begin processing from the latest call to the CHECKPOINT[MANY][X] procedure.

- Identification of the backup process

  The system identifies the process to be affected by the CHECKSWITCH operation from the process's mom field in the process control block (PCB). For named process pairs, this field is automatically set up during the creation of a backup process.

CHILD_LOST_ Procedure

Summary

The CHILD_LOST_ procedure examines a system message to determine whether it indicates that a specified process or process pair has been lost.

When a process receives a system message on $RECEIVE, it can call CHILD_LOST_ to determine whether the message contains information indicating that a particular process has been deleted or has been lost due to a processor or system failure. CHILD_LOST_ reports a loss if any of these are true:

- The connection to a remote system has been lost and the specified process was running in that system.

- A local or remote processor has failed and the specified process was running in that processor.
- A process deletion message has been received for the specified unnamed process, for the specified single named process, or for the entire named process pair of which the specified process is a member.

Syntax for C Programmers

```
#include <cextdecs(CHILD_LOST_ )>

short CHILD_LOST_ ( char *message
                   ,short length
                   ,short *processhandle );
```

Syntax for TAL Programmers

```
status := CHILD_LOST_ ( message:length      ! i:i
                       ,processhandle );    ! i
```

Parameters

status returned value

INT

indicates the result of the check. Valid values are:

0 Process or process pair is not lost

1 (reserved)

2 Parameter error

3 Bounds error

4 Process or process pair is lost

5 System message is not relevant (see *message* parameter, below)

message:*length* input:input

STRING .EXT:ref:*, INT:value

is the status-change message that was received. The *message* process must be exactly *length* bytes long. Relevant messages are:

-2 Local processor down

-5 Process deletion (stop)

-6 Process deletion (abend)

-8 Network status change

-100 Remote processor down

-101 Process deletion

-110 Connection to remote system lost

If any other system message is supplied, a *status* value of 5 is returned.

processhandle input

INT .EXT:ref:10

is the process handle of the process to be checked.

For a check involving a named process pair, it is the process handle of any present or former member of that pair.

Considerations

- CHILD_LOST_ accepts both C-series and D-series format messages. For details about the formats of system messages, see the *Guardian Procedure Errors and Messages Manual*.
- CHILD_LOST_ determines whether a process has been lost by comparing the process or process pair designated in the system message with the process that is specified in the *processhandle* parameter. These tables show the comparison that is made for each system message for each type of process specified by *processhandle*. If the comparison shown in the table is true, the process has been lost.

Message		Local Unnamed Process or Caller's Backup	Local Named Process
-2	Local processor down (unnamed process)	Same processor	N/A
-2	Local processor down (named process)	N/A	Same name
-5	Process deletion (stop)	Same process	Same name and sequence number
-6	Process deletion (abend)	Same process	Same name and sequence number
-101	Process deletion	Same process	Same name and sequence number

Message		Remote Unnamed Process	Remote Named Process

-5	Process deletion (stop)	Same process	Same name and sequence number
-6	Process deletion (abend)	Same process	Same name and sequence number
-8	Network status change	Same node and processor	Same node and all processors down
-100	Remote processor down	Same node and processor	N/A
-101	Process deletion	Same process	Same name and sequence number
-110	Connection to remote node lost	Same node	Same node

Example

```
status := CHILD_LOST_ ( sys^message:length, proc^handle );
```

Related Programming Manual

For programming information about the CHILD_LOST_ procedure, see the *Guardian Programmer's Guide*.

CLOSE Procedure (Superseded by FILE_CLOSE_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development

The CLOSE procedure closes an open file. Closing a file terminates access to the file.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
CALL CLOSE ( filenum                              ! i
           ,[ tape-disposition ] );               ! i
```

Parameters

filenum input

INT:value

is the number of the open file to be closed.

tape-disposition input

INT:value

is one of these values, indicating what tape control action to take:

tape-disposition.<13:15>

0 Rewind and unload; do not wait for completion

1 Rewind and unload; do not wait for completion

2 Rewind, leave online, do not wait for completion

3 Rewind, leave online, wait for completion

4 Do not rewind, leave online

Condition Code Settings

< (CCL) indicates that the file was not open or, for $RECEIVE or the TFILE, there is an outstanding operation using an active transaction.

= (CCE) indicates that the CLOSE was successful.

> (CCG) does not return from CLOSE.

Considerations

- Returning space allocation after closing a file

  Closing a disk file causes the space that is used by the resident file control block to be returned to the system main-memory pool if the disk file is not open concurrently.

A temporary disk file is purged if the file was not open concurrently. Any space that is allocated to that file is made available for other files.

With any file closure, the space allocated to the access control block (ACB) is returned to the system.

- Closing a nowait file

  If a CLOSE is issued for a nowait file that has pending operations, any incomplete operations are canceled. There is no indication as to whether the operation completed or not.

- Labeled tape processing

  If your system has labeled tape processing enabled, all tape actions (as specified by *tape-disposition*) wait for completion.

Messages

- Process close message

  A process can receive a process close system message when it is closed by another process. You can obtain the process ID of the closer in a subsequent call to LASTRECEIVE or RECEIVEINFO. For detailed information of system messages sent to processes, see the *Guardian Procedure Errors and Messages Manual*.

Note. This message is also received if the close is made by the backup process of a process pair. Therefore, a process can expect two of these messages when being closed by a process pair.

Related Programming Manuals

For programming information about the CLOSE file-system procedure, see the *Enscribe Programmer's Guide*.

CLOSE^FILE Procedure

Summary

The CLOSE^FILE procedure closes a specified file.

CLOSE^FILE is a sequential I/O (SIO) procedure and should be used only with files that have been opened by OPEN^FILE.

Syntax for C Programmers

```
#include <cextdecs(CLOSE_FILE)>

short CLOSE_FILE ( { short _near *common-fcb }
                   { short _near *file-fcb   }
                  ,[ short tape-disposition ] );
```

Syntax for TAL Programmers

```
error := CLOSE^FILE ( { common-fcb }             ! i
                      { file-fcb   }             ! i
                     ,[ tape-disposition ] );    ! i
```

Parameters

error returned value

INT

returns either a file-system or a sequential I/O (SIO) procedure error number indicating the outcome of the close. In any case, the file is closed.

If the abort-on-error mode (the default) is in effect, the only possible value for *error* is 0.

common-fcb input

INT:ref:*

indicates that all open files are to be closed if the common file control block (FCB) is passed. If BREAK is owned for any file being closed, it is returned to its previous owner. Note that the first parameter to CLOSE^FILE is either *common-fcb* or *file-fcb*; one or the other can be passed.

file-fcb input

INT:ref:*

identifies the file to be closed if the FCB is passed. If BREAK is owned for the file being closed, it is returned to its previous owner. Note that the first parameter to CLOSE^FILE is either *common-fcb* or *file-fcb*; one or the other can be passed.

tape-disposition input

INT:value

specifies magnetic tape disposition.

tape-disposition.<13:15> denotes:

0 Rewind, unload, do not wait for completion.
1 Rewind, unload, do not wait for completion.
2 Rewind, leave online, do not wait for completion.
3 Rewind, leave online, wait for completion.
4 Do not rewind, leave online.

Other input values result in no error if the file is a tape device; the control action might be unpredictable.

Considerations

- When to use CLOSE^FILE

  Data can be lost if a WRITE^FILE with a count of -1 is not specified or a CLOSE^FILE is not performed against EDIT files or files that are opened with write access and blocking capability before the process is deleted.

- If BREAK is taken, CLOSE^FILE gives BREAK (if owned) to its previous owner.

- For tapes with write access, SIO writes two end-of-file marks (control 2).

- CLOSE^FILE completes all outstanding nowait I/O operations on files that are to be closed.

- If errors occur on more than one file when closing the common FCB, the last encountered error is reported.

- $RECEIVE and CLOSE^FILE

  If the file is $RECEIVE and the user is not handling close messages, SIO waits for a message from each opener. It then replies with either error 45, if read-only access, or error 1, if read/write access, until there are no more openers.

- Errors with CLOSE^FILE

  If you call CLOSE^FILE on the common FCB and if an error is encountered when closing one of the files, the resulting action depends on the setting of ABORT^XFERERR for that file. (ABORT^XFERERR is set by OPEN^FILE or SET^FILE.) If ABORT^XFERERR is true, the process abends. If ABORT^XFERERR is false, a file-system error is returned. In either case, the file in question and all remaining SIO files are closed. If more than one file encounters an error and if they all have ABORT^XFERERR set false, the error returned is that of the last file closed with an error. In all cases where an error is returned by CLOSE^FILE on the common FCB, the program can call CHECK^FILE with the FILE^ERROR operation. This operation can be performed on each file FCB in turn to determine which files encountered an error.

  If CLOSE^FILE returns an error 45 (file is full) for an EDIT file to which data was written, the file will be corrupted because SIO will have been unable to write the appropriate data structures to the end of the file.

Example

```
CALL CLOSE^FILE ( COMMON^FCB );          ! closes all files.
```

Related Programming Manual

For programming information about the CLOSE^FILE procedure, see the *Guardian Programmer's Guide*.

CLOSEALLEDIT Procedure

Summary

The CLOSEALLEDIT procedure closes all open IOEdit files. Calling CLOSEALLEDIT is equivalent to calling the CLOSEEDIT or CLOSEEDIT_ procedure (without the *keep-filenum* parameter) for each file that has been opened by the OPENEDIT or OPENEDIT_ procedure and that has not been closed.

Syntax for C Programmers

```
#include <cextdecs(CLOSEALLEDIT)>

void CLOSEALLEDIT ();
```

Syntax for TAL Programmers

```
CALL CLOSEALLEDIT;
```

Considerations

The CLOSEALLEDIT procedure does not act on any file that has been closed using CLOSEEDIT or CLOSEEDIT_, even if the *keep-filenum* parameter was specified with a nonzero value. In such a case, IOEdit considers the file to be closed even though the file system considers the file to be open and the file number associated with the file is still valid.

Related Programming Manual

For programming information about the CLOSEALLEDIT procedure, see the *Guardian Programmer's Guide*.

CLOSEEDIT Procedure (Superseded by CLOSEEDIT_ Procedure)

Summary

Note. The CLOSEEDIT procedure is supported for compatibility with previous software. For new development, the CLOSEEDIT_ procedure should be used instead.

The CLOSEEDIT procedure closes a specified file that was opened by the OPENEDIT or OPENEDIT_ procedure. The procedure writes to disk any file updates that are still buffered, optionally closes the file through the file system, and finally deallocates all data blocks in the EDIT file segment (EFS) that are associated with the file.

CLOSEEDIT is an IOEdit procedure and can be used only with files that have been opened by OPENEDIT or OPENEDIT_.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
CALL CLOSEEDIT ( filenum                    ! i
                ,[ keep-filenum ] );        ! i
```

Parameters

filenum input

INT:value

is the number that identifies the open file to be closed.

keep-filenum input

INT:value

if supplied and not equal to 0, causes CLOSEEDIT to *not* close the file through the file system, but to perform the rest of its normal operation. This makes it possible to keep the open file number for use in later processing.

Related Programming Manual

For programming information about the IOEdit procedures, see the *Guardian Programmer's Guide*.

CLOSEEDIT_ Procedure

Summary

The CLOSEEDIT_ procedure closes a specified file that was opened by the OPENEDIT or OPENEDIT_ procedure. The procedure writes to disk any file updates that are still buffered, optionally closes the file through the file system, and finally deallocates all data blocks in the EDIT file segment (EFS) that are associated with the file.

CLOSEEDIT_ is an IOEdit procedure and can be used only with files that have been opened by OPENEDIT or OPENEDIT_.

Syntax for C Programmers

```
#include <cextdecs(CLOSEEDIT_ )>

short CLOSEEDIT_ ( short filenum
                 ,[ short keep-filenum ] );
```

Syntax for TAL Programmers

```
error := CLOSEEDIT_ ( filenum                    ! i
                    ,[ keep-filenum ] );         ! i
```

Parameters

error returned value

INT

returns a file-system error number indicating the outcome of the operation.

filenum input

INT:value

is the number that identifies the open file to be closed.

keep-filenum input

INT:value

if supplied and not equal to 0, causes CLOSEEDIT_ to *not* close the file through the file system, but to perform the rest of its normal operation. This makes it possible to keep the open file number for use in later processing.

Related Programming Manual

For programming information about the CLOSEEDIT_ procedure, see the *Guardian Programmer's Guide*.

COMPLETEIOEDIT Procedure

Summary

The COMPLETEIOEDIT procedure informs IOEdit that an outstanding I/O request has finished. Whenever AWAITIO[X] reports the completion of an I/O request on a file that is (or could be) an IOEdit file, you should call COMPLETEIOEDIT. You must supply the output values returned by AWAITIO[X] as the input to COMPLETEIOEDIT.

COMPLETEIOEDIT is an IOEdit procedure and can be used only with files that have been opened by OPENEDIT or OPENEDIT_.

Syntax for C Programmers

```
#include <cextdecs(COMPLETEIOEDIT)>

short COMPLETEIOEDIT ( short filenum
                      ,short count-transferred
                      ,__int32_t tag);
```

- CEXTDECS (through the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
status := COMPLETEIOEDIT ( filenum                    ! i
                         ,count-transferred           ! i
                         ,tag  );                     ! i
```

Parameters

status returned value

INT

returns -1 if *filenum* designates a file being managed by IOEdit; returns 0 otherwise.

filenum input

INT:value

is the number that identifies the open file of interest.

count-transferred input

INT:value

supplies the value of *count-transferred* returned by AWAITIO[X], which gives the count of the number of bytes transferred in the I/O operation.

tag input

INT(32):value

supplies the value of *tag* returned by AWAITIO[X], which gives the application-defined tag that was stored by the system when the I/O operation was initiated.

Related Programming Manual

For programming information about the COMPLETEIOEDIT procedure, see the *Guardian Programmer's Guide*.

COMPRESSEDIT Procedure

Summary

The COMPRESSEDIT procedure copies a specified EDIT file to a new EDIT file that it creates. It fills each block in the new file as much as possible to minimize the number of disk pages used. It then purges the old file and renames the new file to have the name of the old file. The lines in the new file are renumbered if so requested. Upon completion, the new file is open and the current record number is set to -1 (beginning of file). The file number of the new file is returned to the caller.

COMPRESSEDIT is an IOEdit procedure and can only be used with files that have been opened by OPENEDIT or OPENEDIT_.

Syntax for C Programmers

```
#include <cextdecs(COMPRESSEDIT)>

short COMPRESSEDIT ( short *filenum
                    ,short count-transferred
                    ,__int32_t tag);
```

- CEXTDECS (through the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
error := COMPRESSEDIT ( filenum                    ! i,o
                       ,[ start ]                  ! i
                       ,[ increment ] );           ! i
```

Parameters

error returned value

INT

returns a file-system error number indicating the outcome of operation.

filenum input, output

INT .EXT:ref:1

specifies the file number of the open file to be copied into compressed form. It returns the file number of the new file.

start input

INT(32):value

specifies 1000 times the line number of the first line of the new file. You supply this parameter when you want the lines in the new file to be renumbered. If you omit *start*, renumbering still occurs if *increment* is present, in which case the value of *increment* is used for *start*. The possible EDIT line numbers are 0, 0.001, 0.002, ... 99999.999.

increment input

INT(32):value

if present and greater than 0, causes COMPRESSEDIT to renumber the lines in the new file using the incremental value specified. The possible EDIT line numbers are 0, 0.001, 0.002, ... 99999.999. The value of *increment* is 1000 times the value to be added to each successive line number.

If *increment* is not supplied, the line numbers from the original file are used in the new file.

Example

In this example, COMPRESSEDIT copies the specified EDIT file into a new, compressed file in which the line number of the first line is 1 and the line number increment is 1.

```
INT(32) start := 1000D;
INT(32) increment := 1000D;
      .
      .
err := COMPRESSEDIT ( filenumber, start, increment );
```

Related Programming Manual

For programming information about the COMPRESSEDIT procedure, see the *Guardian Programmer's Guide*.

COMPUTEJULIANDAYNO Procedure

Summary

The COMPUTEJULIANDAYNO procedure converts a Gregorian calendar date on or after January 1, 0001, to a Julian day number.

The Julian calendar is the integral number of days since January 1, 4713 B.C. The formal definition of the Julian day states that it starts at 12:00 (noon), Greenwich mean time (GMT).

The Gregorian calendar is the common civil calendar that we use today.

Syntax for C Programmers

```
#include <cextdecs(COMPUTEJULIANDAYNO)>

__int32_t COMPUTEJULIANDAYNO ( short year
                              ,short month
                              ,short day
                              ,[ short _near *error-mask ] );
```

- CEXTDECS (through the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
julian-day-num := COMPUTEJULIANDAYNO ( year                ! i
                                      ,month               ! i
                                      ,day                 ! i
                                      ,[ error-mask ] );   ! o
```

Parameters

julian-day-num returned value

INT(32)

returns the Julian day number or -1 if any input parameter is not within the valid range

year input

INT:value

is the Gregorian year (for example, 1984, 1985, ...). The range for *year* is restricted from 1 through 10000.

month input

INT:value

is the Gregorian month (1-12).

day input

INT:value

is the Gregorian day of the month (1-31).

error-mask output

INT:ref:1

is a bit array in which the first three bits correspond (bit by bit) to *year*, *month*, and *day*, as follows:

```
<0>   year
<1>   month
<2>   day
```

If any one of these bits contains a 1, there is an error. If more than one bit is set, then the combination of elements is bad; which element is actually in error is unknown. For example, 01100000 00000000 is returned for April 31, in which case it is unknown whether April is in error or 31.

Related Programming Manual

For programming information about the COMPUTEJULIANDAYNO procedure, see the *Guardian Programmer's Guide*.

COMPUTETIMESTAMP Procedure

Summary

The COMPUTETIMESTAMP procedure converts a Gregorian (common civil calendar) date and time into a 64-bit Julian timestamp.

Syntax for C Programmers

```
#include <cextdecs(COMPUTETIMESTAMP)>

long long COMPUTETIMESTAMP ( short _near *date-n-time
                           ,[ short _near *errormask ] );
```

Syntax for TAL Programmers

```
ret-timestamp := COMPUTETIMESTAMP ( date-n-time          ! i
                                  ,[ errormask ] );      ! o
```

Parameters

ret-timestamp returned value

FIXED

returns a 64-bit Julian timestamp, computed from *date-n-time*.

date-n-time input

INT:ref:8

is an array containing a date and time of day. The *date-n-time* array has this form:

[0]	the Gregorian year	(for example, 1984, 1985, ...)
[1]	the Gregorian month	(1-12)
[2]	the Gregorian day of the month	(1-31)
[3]	the hour of the day	(0-23)
[4]	the minute of the hour	(0-59)
[5]	the second of the minute	(0-59)
[6]	the millisecond of the second	(0-999)
[7]	the microsecond of the millisecond	(0-999)

The range of the year is restricted from 1 through 10000.

errormask output

INT:ref:1

is a bit array that indicates any error in the *date-n-time* parameter. The *errormask* parameter checks each element of *date-n-time* for validity. If *errormask* is omitted, *date-n-time* is not checked.

An error is indicated if any of these bits contains a 1. The *errormask* bits are:

<0>	year
<1>	month
<2>	day

<3> hour of day
<4> minute of hour
<5> second of minute
<6> millisecond of second
<7> microsecond of millisecond

If more than one bit is set, the combination of elements is bad; which element is actually in error is unknown. For example, 01100000 00000000 is returned for April 31, in which case it is unknown whether April is in error or 31.

Considerations

- A 64-bit Julian timestamp is based on the Julian Date. It is a quantity equal to the number of microseconds since January 1, 4713 B.C., 12:00 (noon) Greenwich mean time (Julian proleptic calendar). This timestamp can represent either Greenwich mean time, local standard time, or local civil time. There is no way to examine a Julian timestamp and determine which of the three times it represents.
- Procedures that work with the 64-bit Julian timestamp are COMPUTETIMESTAMP, CONVERTTIMESTAMP, INTERPRETTIMESTAMP, JULIANTIMESTAMP, and SETSYSTEMCLOCK.
- For a more complete description of 48-bit and 64-bit timestamps, see the TIMESTAMP or JULIANTIMESTAMP procedure.

Related Programming Manual

For programming information about the COMPUTETIMESTAMP procedure, see the *Guardian Programmer's Guide*.

CONFIG_GETINFO_BYLDEV_ Procedure (G-Series and H-Series RVUs Only)

CONFIG_GETINFO_BYNAME_ Procedure (G-Series and H-Series RVUs Only)

Summary

The CONFIG_GETINFO_BYLDEV_ and CONFIG_GETINFO_BYNAME_ procedures obtain the logical and physical attributes of a device on a G-series RVU. To specify the device by logical device number, use the CONFIG_GETINFO_BYLDEV_ procedure. To specify the device by name, use the CONFIG_GETINFO_BYNAME_ procedure.

The CONFIG_GETINFO_BYLDEV_ procedure is provided to simplify migration from earlier hardware. This procedure does not return information from subtype 30 processes. For new development, use the CONFIG_GETINFO_BYNAME_ procedure.

Syntax for C Programmers

Note. In the TNS/E environment, the CEXTDECS file uses the *int* data type for 32-bit values. This is a change from the TNS and TNS/R environments where CEXTDECS uses the *long* data type for 32-bit values.

```
#include <cextdecs(CONFIG_GETINFO_BYLDEV_)>


__int32_t CONFIG_GETINFO_BYLDEV2_ ( __int32_t ldevnum
                                  , short common-info-maxlen
                                  , short *common-info-len
                                  , char *specific-info
                                  , short specific-info-maxlen
                                  , short *specific-info-len
                                  , __int32_t timeout
                                  , __int32_t *error-detail  );
```

```
#include <cextdecs(CONFIG_GETINFO_BYNAME_)>

__int32_t CONFIG_GETINFO_BYNAME_ ( char *devname
                                 , short length
                                 , short *common-info
                                 , short common-info-maxlen
                                 , short *common-info-len
                                 , char *specific-info
                                 , short specific-info-maxlen
                                 , short *specific-info-len
                                 , __int_32_t  timeout
                                 , __int32_t *error-detail  );
```

Syntax for TAL Programmers

```
error := CONFIG_GETINFO_BYLDEV_ (
              ldevnum                                  ! i
             ,common-info                              ! o
             ,common-info-maxlen                       ! i
             ,common-info-len                          ! o
             ,specific-info:specific-info-maxlen       ! o:i
             ,specific-info-len                        ! o
             ,timeout                                  ! i
             ,error-detail  );                         ! o
```

```
error := CONFIG_GETINFO_BYNAME_ (
              devname:length                           ! i
             ,common-info                              ! o
             ,common-info-maxlen                       ! i
             ,common-info-len                          ! o
             ,specific-info:specific-info-maxlen       ! o:i
             ,specific-info-len                        ! o
             ,timeout                                  ! i
             ,error-detail  );                         ! o
```

Parameters

error returned value

INT(32)

indicates the outcome of the operation. It returns one of these values:

0D Information was successfully returned.

1D Either the device or the process simulating a device detected a file-system error; *error-detail* contains a file-system error number. If *error-detail* is 14D, the device was not found. If *error-detail* is 40D, the I/O subsystem did not respond within the *timeout* specified.

2D Parameter error; *error-detail* contains the number of the first parameter found to be in error, where 1D designates the first parameter on the left.

3D Bounds error; *error-detail* contains the number of the first parameter found to be in error, where 1D designates the first parameter on the left.

4D Either the device or the process simulating a device detected an error; *error-detail* contains the error number returned by the device. If *error-detail* is -1D, the returned information is invalid.

ldevnum (CONFIG_GETINFO_BYLDEV_ only) input

INT(32):value

specifies the logical device number of the device for which information is requested. The logical device number of a device can change whenever a device is configured or the system is loaded. Some I/O subsystems do not have a logical device number.

devname:*length* (CONFIG_GETINFO_BYNAME_ only) input: input

STRING .EXT:ref:*, INT:value

specifies the name of the device for which information is requested. The value of *devname* must be a local name (that is, it must not include a system name) and can have qualifiers.

The devname parameter must be exactly *length* bytes long.

common-info output

INT .EXT:ref:(ZSYS^DDL^CONFIG^GETINFO)

if *error* is 0D and if *common-info-maxlen* is not 0, points to a buffer that returns a set of logical attributes for the specified device. For information on the values in the buffer, see “Structure Definition for *common-info*.”

common-info-maxlen input

INT:value

specifies the length in bytes of the buffer pointed to by *common-info*. If the buffer length is too short for the full set of device attributes, the procedure sets *error* to 2D, sets *error-detail* to 3D, and does not return any information on the specified device.

common-info-len output

INT .EXT:ref:1

returns the actual length in bytes of the buffer pointed to by *common-info*.

specific-info:specific-info-maxlen output:input

STRING .EXT:ref:*, INT:value

if present and if *specific-info-maxlen* is not 0, points to a buffer that returns a set of physical device attributes obtained from the I/O subsystem that supports the specified device. The attribute values are returned in a structure that is defined by the I/O subsystem. See I/O Subsystems That Use the specific_info Parameter on page 3-89 for a detailed description of the structure of this buffer.

specific-info-maxlen specifies the length in bytes of the buffer pointed to by *specific-info*. If the buffer length is too short for the full set of device attributes, the procedure returns as many values as will fit in the buffer.

If this parameter pair is present, *specific-info-len* must also be present.

specific-info-len output

INT .EXT:ref:1

returns the actual length in bytes of the buffer pointed to by *specific-info*. If *specific-info-len* is greater than *specific-info-maxlen*, then *specific-info* does not contain all the available data from the I/O subsystem.

This parameter must be present if *specific-info* is present.

timeout input

INT(32):value

specifies how many hundredths of a second the procedure should wait for a response from the I/O subsystem. The maximum value is 2147483647. The default value is 6000D (one minute). A value of -1D causes the procedure to wait indefinitely.

error-detail output

INT(32) .EXT:ref:1

for some returned errors, contains additional information. See *error*, earlier in this subsection.

Structure Definition for *common-info*

The *common-info* parameter points to a buffer that returns a set of logical attributes for a specified device.

In the TAL ZSYSTAL file, the structure of the buffer that the *common-info* parameter points to is defined below.

```
STRUCT       ZSYS^DDL^CONFIG^GETINFO^DEF (*)
?IF PTAL
FIELDALIGN (SHARED2)
?ENDIF PTAL;
   BEGIN
   INT          Z^EYECATCHER;
   INT          Z^VERSION;
   STRUCT       Z^NSK^NODENAME;
      BEGIN STRING BYTE [0:7]; END;
   STRUCT       Z^NSK^DEVICENAME;
      BEGIN STRING BYTE [0:7]; END;
   STRUCT       Z^QUALIFIER1;
      BEGIN STRING BYTE [0:7]; END;
   STRUCT       Z^QUALIFIER2;
      BEGIN STRING BYTE [0:7]; END;
   INT(32)      Z^LDEV^NUMBER;
   INT          Z^DEVICE^RECORD^SIZE;
   INT          Z^DEVICE^TYPE;
   INT          Z^DEVICE^SUBTYPE;
   INT          Z^LOGICAL^STATUS;
   INT          Z^CONFIG^NAME^LEN;
   STRUCT       Z^CONFIG^NAME;
      BEGIN STRING BYTE [0:63]; END;
   INT          Z^SUBSYS^MANAGER^LEN;
   STRUCT       Z^SUBSYS^MANAGER;
      BEGIN STRING BYTE [0:47]; END;
   STRUCT       Z^PRIMARY^PHANDLE;
      BEGIN
      STRUCT       Z^DATA;
         BEGIN
         STRING       ZTYPE;
         FILLER       19;
         END;
      INT          Z^WORD[0:9] = Z^DATA;
      STRUCT       Z^BYTE = Z^DATA;
         BEGIN STRING BYTE [0:19]; END;
      END;
   STRUCT       Z^BACKUP^PHANDLE;
      BEGIN
      STRUCT       Z^DATA;
         BEGIN
         STRING       ZTYPE;
         FILLER       19;
         END;
      INT          Z^WORD[0:9] = Z^DATA;
      STRUCT       Z^BYTE = Z^DATA;
         BEGIN STRING BYTE [0:19]; END;
      END;
   STRUCT       Z^RESERVED^1;
      BEGIN STRING BYTE [0:19]; END;
   END;
```

In the C `zsysc` file, the structure of the buffer that the *`common-info`* parameter points to is defined below.

```
#pragma fieldalign shared2 __zsys_ddl_config_getinfo
typedef struct __zsys_ddl_config_getinfo
{
   short                              z_eyecatcher;
   short                              z_version;
   char                               z_nsk_nodename[8];
   char                               z_nsk_devicename[8];
   char                               z_qualifier1[8];
   char                               z_qualifier2[8];
   long                               z_ldev_number;
   short                              z_device_record_size;
   short                              z_device_type;
   short                              z_device_subtype;
   short                              z_logical_status;
   short                              z_config_name_len;
   char                               z_config_name[64];
   short                              z_subsys_manager_len;
   char                               z_subsys_manager[48];
   zsys_ddl_phandle_def               z_primary_phandle;
   zsys_ddl_phandle_def               z_backup_phandle;
   char                               z_reserved_1[20];
} zsys_ddl_config_getinfo_def;

#pragma section zsys_ddl_phandle
#pragma fieldalign shared2 __zsys_ddl_phandle
typedef struct __zsys_ddl_phandle
{
   union
   {
      struct
      {
         signed char                        ztype;
         char                               filler_0[19];
      } z_data;
      short                           z_word[10];
      char                            z_byte[20];
   } u_z_data;
} zsys_ddl_phandle_def;
```

`Z^EYECATCHER`

identifies the structure and is helpful in debugging.

This TAL literal is defined in the ZSYSTAL file. Literals in the `zsysc` file, for C programs, are the same as those for TAL except that they contain the underscore (_) character instead of the circumflex (^) character.

Name (ZSYS^VAL^)	Value	ASCII Value	Description
CONFIG^GT^EYCATCHER	17225	“CI”	Flag for debugging

Z^VERSION

identifies the version of the ZSYS^DDL^CONFIG^GETINFO structure.

This TAL literal is defined in the ZSYSTAL file. Literals in the zsysc file, for C programs, are the same as those for TAL except that they contain the underscore (_) character instead of the circumflex (^) character.

Name (ZSYS^VAL^)	Value	Description
CONFIG^GI^VERSION	1	The current version of the structure

Z^NSK^NODENAME

is the node of the device name returned in Z^NSK^DEVICENAME.

Z^NSK^DEVICENAME

is the local name (that is, a name that does not include a node name) of the device whose attributes are being returned. This name has no qualifiers. Z^NSK^NODENAME returns the node name, and Z^QUALIFIER1 and Z^QUALIFIER2 return any qualifiers.

Z^QUALIFIER1

is the first qualifier name subordinate to the device name returned in Z^NSK^DEVICENAME. For example, #Q1 is the first qualifier of the terminal process named $TERM.#Q1.Q2.

Z^QUALIFIER2

is the second qualifier name subordinate to the device name returned in Z^NSK^DEVICENAME. For example, Q2 is the second qualifier of the terminal process named $TERM.#Q1.Q2.

Z^LDEV^NUMBER

is the logical device number of the device whose attributes are being returned. The logical device number of a device can change whenever a device is configured or the system is loaded. Some I/O subsystems do not have a logical device number.

Z^DEVICE^RECORD^SIZE

is the record size of the device.

Z^DEVICE^TYPE

is the device type of the device. See Appendix A, Device Types and Subtypes for a list of device types.

Z^DEVICE^SUBTYPE

is the device subtype of the device. See Appendix A, Device Types and Subtypes for a list of device subtypes.

Z^LOGICAL^STATUS

is the logical status of the device. This TAL literals are defined in the ZCOMTAL file. Literals in the zcomc file, for C programs, are the same as those for TAL except that they contain the underscore (_) character instead of the circumflex (^) character.

Name (ZCOM^VAL^)	Value	Indicates the I/O subsystem is
SUMSTATE^ABORTING	0	aborting
SUMSTATE^DEF	1	defined
SUMSTATE^DIAG	2	running diagnostics
SUMSTATE^STARTED	3	started
SUMSTATE^STARTING	4	starting
SUMSTATE^STOPPED	5	stopped
SUMSTATE^STOPPING	6	stopping
SUMSTATE^SUSP	7	suspended
SUMSTATE^UNKWN	8	in an unknown state
SUMSTATE^SUSPENDING	9	suspending
SUMSTATE^SERVICE	10	being serviced

Z^CONFIG^NAME^LEN

is the length of the configured name of the device, which is in Z^CONFIG^NAME.

Z^CONFIG^NAME

is the configured name of the device.

Z^SUBSYS^MANAGER^LEN

is the length of the name of the Guardian subsystem manager process, which is in Z^SUBSYS^MANAGER.

Z^SUBSYS^MANAGER

is the name of the Guardian subsystem manager process.

Z^PRIMARY^PHANDLE

is the process handle of the primary I/O subsystem that owns the device.

Z^BACKUP^PHANDLE

is the process handle of the backup I/O subsystem that owns the device.

Z^RESERVED^1

is reserved.

Considerations

- It is possible for CONFIG_GETINFO_BYLDEV_ to return an *error* value of 0D (information successfully returned) while the I/O subsystem reports an error in the Z^LOGICAL^STATUS field of the returned buffer. In that case, the *error* value of 0D indicates that communication with the I/O subsystem was successful, while the I/O subsystem logical status value reflects the status of the I/O subsystem.
- Searching logical devices

  To perform a search of logical devices, call the file-name inquiry procedures FILENAME_FINDSTART_, FILENAME_FINDNEXT_, and FILENAME_FINDFINISH_.

I/O Subsystems That Use the *specific_info* Parameter

The *specific_info* parameter contains information passed back from the subsystem called by CONFIG_GETINFO_BYNAME (and the other CONFIG_GETINFO procedures). The structure of this parameter is different depending on which subsystem is called. The CONFIG_GETINFO_BYNAME procedure supports these subsystems:

- Storage subsystem
- ServerNet Lan Subsystem Access (SLSA) Subsystem
- ServerNet Wide Area Network (SWAN)

Storage Subsystem

The structure returned by the storage subsystem in the *specific_info* parameter of the reply consists of a header field followed by a number of path info fields.

The structure is specific to the type of device being described; for example, the information for a magnetic disk or SCSI-IOP device contains SCSI-specific information, and the path for a tape device contains tape-specific information, and so on.

The structure for magnetic disk and SCSI-IOP devices is defined in TAL as follows:

```
STRUCT       SCSI^DEVICE^INFO^DEF (*) FIELDALIGN (SHARED2);
   BEGIN
   STRUCT       SPECIFIC^HDR;
      BEGIN
      INT          VERSION;
      INT          MAX^NUM^PATHS;
      INT          PRIMARY^SUBTYPE;
      INT          MIRROR^SUBTYPE;
      END;
   INT          AUDITED;
   INT          DEMOUNTABLE;
   INT          RESERVE1;
   INT          FLAGS;
   STRUCT       PATH^INFO[0:3];
      BEGIN
      STRUCT       STANDARD;
         BEGIN
         INT          CONFIGURED;
         INT          INUSE;
         INT          STATE;
         INT          RESERVED;
         INT(32)      GROUP;
         INT(32)      MODULE;
         INT(32)      SLOT;
         INT(32)      SUBDEVNUM;
         INT(32)      FABRIC;
         FILLER       4;
         STRUCT       SACNAME;
            BEGIN STRING BYTE [0:63]; END;
         INT(32)      PRIMARYCPU;
         INT(32)      BACKUPCPU;
         END;
      FIXED        SCSILUN;
      FIXED        SCSITARGET;
      END;
      FIXED        PORT^NAME[0:3]; /*New field*/
   END;
```

The structure for tape devices is defined in TAL as follows:

```
STRUCT       SCSI^TAPE^DEVICE^INFO^DEF (*) FIELDALIGN (SHARED2);
   BEGIN
   STRUCT       SPECIFIC^HDR;
      BEGIN
      INT          VERSION;
      INT          MAX^NUM^PATHS;
      INT          PRIMARY^SUBTYPE;
      INT          MIRROR^SUBTYPE;
      END;
   STRUCT       PATH^INFO;
      BEGIN
      STRUCT       STANDARD;
         BEGIN
         INT          CONFIGURED;
         INT          INUSE;
         INT          STATE;
         INT          RESERVED;
         INT(32)      GROUP;
         INT(32)      MODULE;
         INT(32)      SLOT;
         INT(32)      SUBDEVNUM;
         INT(32)      FABRIC;
         FILLER       4;
         STRUCT       SACNAME;
            BEGIN STRING BYTE [0:63]; END;
         INT(32)      PRIMARYCPU;
         INT(32)      BACKUPCPU;
         END;
      FIXED        SCSILUN;
      FIXED        SCSITARGET;
      END;
      FIXED        PORT^NAME[0:3]; /*New field*/
   END;
```

These parameter descriptions apply to both structures above, with the exception of 5 parameters that are used for magnetic disk and SCSI-IOP devices only.

`version`

Structure version

`max_num_paths`

The maximum number of paths for the device type.

Device Type	max_num_paths
disk	4
tape	1
open-scsi	2

`primary_subtype`

Subtype of the magnetic disk located on the primary volume. For more information about device subtypes, see Appendix A, Device Types and Subtypes.

`mirror_subtype`

Subtype of the magnetic disk located on the mirror volume. For more information about device subtypes, see Appendix A, Device Types and Subtypes.

`audited`

(For magnetic disk and SCSI-IOP devices only.) This volume is currently audited for TMF

`demountable`

(For magnetic disk and SCSI-IOP devices only.) Always 1 - the volume can be always be removed

`reserve1`

(For magnetic disk and SCSI-IOP devices only.) (not used)

`flags`

(For magnetic disk and SCSI-IOP devices only.)

<15> = 1 Volume is in SOFTDOWN state
<14> = 1 Backup DP2 is in SOFTDOWN state

`path^info`

(For magnetic disk and SCSI-IOP devices only.) Array of STRUCT STORAGE_PATH_INFO_HEADER_DEF(*), one for each path

`configured`

Equal to -1 if the path is configured, equal to 0 if the path is not configured. Devices such as terminals and tape drives have only one path configured; disks can have two or four paths configured.

`inuse`

Equal to -1 if the path is currently in use by the IOP that owns the device, equal to 0 if it is not.

`state`

The current state of the path. If the device has only one path, then the state of the device is the state of the path. Valid state values are:

Value	Description
0	UP
1	DOWN
2	SPECIAL

3	MOUNT
4	REVIVE
5	(reserved)
6	EXERCISE
7	EXCLUSIVE
8	HARD DOWN
9	UNKNOWN

`group`

All objects accessible to a pair of service processors in a system enclosure. In a NonStop S-series server, a group includes all components in a system enclosure. The valid range is from 1 through 999.

`module`

A set of components that shares a hardware interconnection. A module is a subset of a group. It is contained in a system enclosure, and contains one or more slots. In a NonStop S-series server, there is exactly one module in a group. The valid range is from 1 through 99.

`slot`

A physical, labeled space in a module in which a CRU can be installed. The valid range is from 1 through 999.

`subDevNum`

The subtype of the device. For more information about device subtypes see Appendix A, Device Types and Subtypes.

`fabric`

X, Y, or both. These codes correspond to the possible fabric types:

Value	Fabric
0	X
1	Y

`sacName`

The name of the ServerNet addressable controller (SAC) located in the PMF CRU or IOMF CRU.

`scsiLun`

The logical unit number (LUN) of the Open SCSI device.

`scsiTargetId`

The SCSI ID of the Open SCSI device. This is the address used by the Open SCSI I/O process to access the device.

More information about the definition of the *specific_info* field for the Storage subsystem can be found in the Guardian file `$SYSTEM.ZSPIDEF.ZSTOTAL`.

ServerNet Wide Area Network (SWAN)

Below is the structure definition for the *specific_info* parameter when returned by a device that uses ServerNet wide area network (SWAN) connectivity. The structure is defined by the wide area network (WAN) subsystem. Devices for many HP communication subsystems, including AM3270, ATP, CP6100, Envoy and Envoy ACP/XF, Expand, SNAX/XF and SNAX/APN, TR3271, and X25AM support SWAN connectivity in NonStop S-series servers. (The structure definition below applies only to the subsystems identified here.) Note that if an Expand or a SNAX device does not use SWAN, null values are returned in this structure.

This structure definition can be found in the TAL ZWANTAL file.

```
STRUCT ZWAN^DDL^EXIOADDR^DEF (*)
 ?IF PTAL
 FIELDALIGN (SHARED2)
 ?ENDIF PTAL
 ;
    BEGIN
    INT          Z^PATYPE;
    INT          Z^CHNL;
    INT          Z^CTLR;
    INT          Z^CLIPNUM = Z^CTLR;
    INT          Z^UNIT;
    INT          Z^LINENUM = Z^UNIT;
    INT          Z^CPU;
    STRUCT       Z^TRACKID;
       BEGIN
       STRUCT       Z^C;
          BEGIN STRING BYTE [0:5]; END;
       STRUCT       Z^S = Z^C;
          BEGIN
          INT          Z^I[0:2];
          END;
       STRING       Z^B[0:5] = Z^C;
       END;
    INT(32)      Z^IPADDRESS;
    STRUCT       Z^IPADDRSTG = Z^IPADDRESS;
       BEGIN
       STRUCT       Z^C;
          BEGIN STRING BYTE [0:3]; END;
       STRUCT       Z^S = Z^C;
          BEGIN
          INT          Z^I[0:1];
          END;
       STRING       Z^B[0:3] = Z^C;
       END;
    END;
```

Z^PATYPE

is the physical adapter type. For SWAN support, Z^PATYPE has a value of 4.

Z^CHNL

is reserved.

Z^CTLR

is reserved. It is redefined by Z^CLIPNUM.

Z^CLIPNUM

is the clip number that identifies which clip number is being used. The valid range for the clip number is 1 through 3.

Z^UNIT

is reserved. It is redefined by Z^LINENUM.

Z^LINENUM

identifies the line number (0 or 1) on a clip that is being used.

Z^CPU

is the processor that is currently being used to run the LINE. The LINE is a Subsystem Programmatic Interface (SPI) object in a subsystem.

Z^TRACKID

is a six-character string number that identifies the SWAN box.

Z^IPADDRESS

is the IP address that is currently being used to access the SWAN.

CONFIG_GETINFO_BYLDEV2_ Procedure (G-Series and H-Series RVUs Only)

CONFIG_GETINFO_BYNAME2_ Procedure (G-Series and H-Series RVUs Only)

Summary

The CONFIG_GETINFO_BYLDEV2_ and CONFIG_GETINFO_BYNAME2_ procedures obtain the logical and physical attributes of a device on a G-series RVU. To specify the device by logical device number, use the CONFIG_GETINFO_BYLDEV2_ procedure. To specify the device by name, use the CONFIG_GETINFO_BYNAME2_ procedure.

The CONFIG_GETINFO_BYLDEV2_ and CONFIG_GETINFO_BYNAME2_ procedures are variants of CONFIG_GETINFO_BYLDEV_ and CONFIG_GETINFO_BYNAME_. The CONFIG_GETINFO_BYLDEV2_ and CONFIG_GETINFO_BYNAME2_ procedures allow the caller to specify device names that do not conform to Guardian file-name formats as required by some communication devices.

Syntax for C Programmers

Note. In the TNS/E environment, the CEXTDECS file uses the *int* data type for 32-bit values. This is a change from the TNS and TNS/R environments where CEXTDECS uses the *long* data type for 32-bit values.

```
#include <cextdecs(CONFIG_GETINFO_BYLDEV2_)>

__int32_t CONFIG_GETINFO_BYLDEV2_ ( __int32_t ldevnum
                                  , short *common-info
                                  , short common-maxlen
                                  , short *common-len
                                  , char *specific-info ]
                                  , short specific-maxlen ]
                                  , short *specific-len ]
                                , __int32_t timeout ]
                                , __int32_t *error-detail ] );
```

```
#include <cextdecs(CONFIG_GETINFO_BYNAME2_)>

__int32_t CONFIG_GETINFO_BYNAME2_ ( char *devname
                                  , short length
                                  , short *common-info
                                  , short common-maxlen
                                  , short *common-len
                                  , char *specific-info
                                  , short specific-maxlen
                                  , short *specific-len
                                , __int32_t  timeout
                                , __int32_t *error-detail  );
```

Syntax for TAL Programmers

```
error := CONFIG_GETINFO_BYLDEV2_ (
              ldevnum                                  ! i
             ,common-info                              ! o
             ,common-maxlen                            ! i
             ,common-len                               ! o
             ,specific-info:specific-maxlen            ! o:i
             ,specific-len                             ! o
             ,timeout                                  ! i
             ,error-detail  );                         ! o
```

```
error := CONFIG_GETINFO_BYNAME2_ (
              devname:length                           ! i
             ,common-info                              ! o
             ,common-maxlen                            ! i
             ,common-len                               ! o
             ,specific-info:specific-maxlen            ! o:i
             ,specific-len                             ! o
             ,timeout                                  ! i
             ,error-detail  );                         ! o
```

Parameters

error returned value

INT(32)

indicates the outcome of the operation. It returns one of these values:

0D Device found and data is returned. The value of *error-detail* is set to zero. Only if error is returned as zero will there be any meaningful data in *common-info* or *specific-info*.

1D Device or subtype 30 process returned an error. The error is reported in *error-detail*. An *error-detail* value of 40 indicates that the I/O process did not respond within the timeout interval. An *error-detail* value of 14 indicates that the device was not found.

2D Required parameter is invalid. The value of *error-detail* is set to the ordinal number of the invalid parameter.

3D Bounds error; a reference parameter contained an invalid address. The value of *error-detail* is set to the ordinal number of the invalid parameter.

4D Device returned error or invalid data to the inquiry. If *Error-detail* is -1, then the device returned zero and the response is invalid. Otherwise, *error-detail* is the value of the error returned to the inquiry by the device.

ldevnum (CONFIG_GETINFO_BYLDEV2_ only) input

INT(32):value

specifies the logical device number of the device for which information is requested.

devname:*length* (CONFIG_GETINFO_BYNAME2_ only) input: input

STRING .EXT:ref:*, INT:value

specifies the name of the device to look up. The *devname* parameter and the *length* parameter are required. The *length* parameter is the length of the string in bytes.

common-info output

INT .EXT:ref:(ZSYS^DDL^CONFIG^GETINFO2)

specifies a pointer to a buffer that will hold the result of the call. The structure is always returned, if *error* has zero value.

common-maxlen input

INT:value

specifies the length in bytes of the buffer pointed to by *common-info*. This value must be greater than or equal to the size of *common-info*; otherwise, *error* is set to 2 and *error-detail* is set to 3.

common-len output

INT .EXT:ref:1

number of bytes returned by *common-info*. If the value returned by this procedure is nonzero, this parameter will be returned as zero.

specific-info:specific-maxlen output:input

STRING .EXT:ref:1, INT:value

specifies an area where the device-specific information will be returned. If *specific-info* is zero or if *specific-maxlen* is zero, then no device-specific data is returned. Each device type or subtype might return a different set of data. See I/O Subsystems That Use the specific_info Parameter on page 3-89 for a detailed description of the structure of this buffer.

If the returned device-specific information is too large to fit into the buffer, the actual data is truncated to fit; however, *specific-len* is set to reflect the number of bytes that would have been returned if the buffer had been large enough.

specific-len output

INT .EXT:ref:1

if *specific-info* is not 0 and *specific-maxlen* is greater than zero, this parameter is returned if its value is not 0. The value returned reflects the full size of the device-specific information, even if it is larger than the *specific-maxlen* value. The actual number of bytes copied into *specific-info* is the smaller of *specific-maxlen* and *specific-len* bytes. If the value returned by this procedure is nonzero, this parameter is returned as zero.

timeout input

INT(32):value

specifies how many hundredths of a second the procedure should wait for a response from the device. If zero is passed, the timeout value is set to its default (6000D or 1 minute). If -1D is passed, the procedure does not time out.

error-detail output

INT(32) .EXT:ref:1

If *error-detail* is not 0, then an error-dependent value is returned.

Structure Definition for *common-info*

The *common-info* parameter points to a buffer that returns a set of logical attributes for a specified device.

In the TAL ZSYSTAL file, the structure of the buffer that the *common-info* parameter points to is defined as follows.

```
STRUCT    ZSYS^DDL^CONFIG^GETINFO2^DEF (*) FIELDALIGN
(SHARED2);
   BEGIN
   INT          Z^EYECATCHER;
   INT          Z^VERSION;
   STRUCT       Z^NSK^NODENAME;
      BEGIN STRING BYTE [0:7]; END;
   INT(32)      Z^LDEV^NUMBER;
   INT          Z^DEVICE^RECORD^SIZE;
   INT          Z^DEVICE^TYPE;
   INT          Z^DEVICE^SUBTYPE;
   INT          Z^LOGICAL^STATUS;
   INT          Z^CONFIG^NAME^LEN;
   STRUCT       Z^CONFIG^NAME;
      BEGIN STRING BYTE [0:63]; END;
   INT          Z^SUBSYS^MANAGER^LEN;
   STRUCT       Z^SUBSYS^MANAGER;
      BEGIN STRING BYTE [0:47]; END;
   STRUCT       Z^PRIMARY^PHANDLE;
      BEGIN
      STRUCT       Z^DATA;
         BEGIN
         STRING       ZTYPE;
         FILLER       19;
         END;
      INT          Z^WORD[0:9] = Z^DATA;
      STRUCT       Z^BYTE = Z^DATA;
         BEGIN STRING BYTE [0:19]; END;
      END;
   STRUCT       Z^BACKUP^PHANDLE;
      BEGIN
      STRUCT       Z^DATA;
         BEGIN
         STRING       ZTYPE;
         FILLER       19;
         END;
      INT          Z^WORD[0:9] = Z^DATA;
      STRUCT       Z^BYTE = Z^DATA;
         BEGIN STRING BYTE [0:19]; END;
      END;
   STRUCT       Z^RESERVED^1;
      BEGIN STRING BYTE [0:19]; END;
   INT          Z^DEVNAME^OFF;
   INT          Z^DEVNAME^LEN;
   STRUCT       Z^DEVNAME^DATA;
      BEGIN STRING BYTE [0:63]; END;
   END;
```

In the C zsysc file, the structure of the buffer that the *common-info* parameter points to is defined as follows.

```
#pragma fieldalign shared2 __zsys_ddl_config_getinfo2
typedef struct __zsys_ddl_config_getinfo2
{
   short                              z_eyecatcher;
   short                              z_version;
   char                               z_nsk_nodename[8];
   long                               z_ldev_number;
   short                              z_device_record_size;
   short                              z_device_type;
   short                              z_ldev_subtype;
   short                              z_logical_status;
   short                              z_config_name_len;
   char                               z_config_name[64];
   short                              z_subsys_manager_len;
   char                               z_subsys_manager[48];
   zsys_ddl_phandle_def               z_primary_phandle;
   zsys_ddl_phandle_def               z_backup_phandle;
   char                               z_reserved_1[20];
   short                              z_devname_off;
   short                              z_devname_len;
   char                               z_devname_data[64];
} zsys_ddl_config_getinfo2_def;

#pragma section zsys_ddl_phandle
#pragma fieldalign shared2 __zsys_ddl_phandle
typedef struct __zsys_ddl_phandle
{
   union
   {
      struct
      {
         signed char                     ztype;
         char                            filler_0[19];
      } z_data;
      short                           z_word[10];
      char                            z_byte[20];
   } u_z_data;
} zsys_ddl_phandle_def;
```

Z^EYECATCHER

identifies the structure and is helpful in debugging.

This TAL literal is defined in the ZSYSTAL file. Literals in the zsysc file, for C programs, are the same as those for TAL except that they contain the underscore (_) character instead of the circumflex (^) character.

Name (ZSYS^VAL^)	Value	ASCII Value	Description
CONFIG^GT^EYCATCHER	17225	“DI”	Flag for debugging

Z^VERSION

identifies the version of the ZSYS^DDL^CONFIG^GETINFO structure.

This TAL literal is defined in the ZSYSTAL file. Literals in the zsysc file, for C programs, are the same as those for TAL except that they contain the underscore (_) character instead of the circumflex (^) character.

Name (ZSYS^VAL^)	Value	Description
CONFIG^GI^VERSION	2	The current version of the structure

Z^NSK^NODENAME

is the node of the device name returned in Z^DEVNAME^DATA.

Z^LDEV^NUMBER

is the logical device number of the device whose attributes are being returned. The logical device number could be set to -1. The logical device number of a device can change whenever a device is configured or the system is loaded. Some I/O subsystems do not have a logical device number.

Z^DEVICE^RECORD^SIZE

is the record size of the device.

Z^DEVICE^TYPE

is the type of the device. See Appendix A, Device Types and Subtypes for a list of device types.

Z^DEVICE^SUBTYPE

is the subtype of the device. See Appendix A, Device Types and Subtypes for a list of device subtypes.

Z^LOGICAL^STATUS

is the logical status of the device. These TAL literals are defined in the ZCOMTAL file. Literals in the zcomc file, for C programs, are the same as those for TAL except that they contain the underscore (_) character instead of the circumflex (^) character.

Name (ZCOM^VAL^)	Value	Indicates the I/O subsystem is
SUMSTATE^ABORTING	0	Terminating abnormally
SUMSTATE^DEF	1	Defined
SUMSTATE^DIAG	2	Running diagnostics
SUMSTATE^STARTED	3	Started
SUMSTATE^STARTING	4	Starting
SUMSTATE^STOPPED	5	Stopped

Name (ZCOM^VAL^)	Value	Indicates the I/O subsystem is
SUMSTATE^STOPPING	6	Stopping
SUMSTATE^SUSP	7	Suspended
SUMSTATE^UNKWN	8	In an unknown state
SUMSTATE^SUSPENDING	9	Suspending
SUMSTATE^SERVICE	10	Being serviced

`Z^CONFIG^NAME^LEN`

is the length of the configured name of the device, which is in Z^CONFIG^NAME.

`Z^CONFIG^NAME`

is the configured name of the device.

`Z^SUBSYS^MANAGER^LEN`

is the length of the name of the Guardian subsystem manager process, which is in Z^SUBSYS^MANAGER.

`Z^SUBSYS^MANAGER`

is the name of the Guardian subsystem manager process.

`Z^PRIMARY^PHANDLE`

is the process handle of the primary I/O subsystem that owns the device.

`Z^BACKUP^PHANDLE`

is the process handle of the backup I/O subsystem that owns the device.

`Z^RESERVED^1`

is reserved.

`Z^DEVNAME^OFF`

is the offset of Z^DEVNAME^DATA from beginning of the `common-info` parameter.

`Z^DEVNAME^LEN`

is the length of the device name in bytes. The maximum length is 64.

`Z^DEVNAME^DATA`

is the full device name.

Considerations

- It is possible for CONFIG_GETINFO_BYLDEV2_ to return an *error* value of 0D (information successfully returned) while the I/O subsystem reports an error in the Z^LOGICAL^STATUS field of the returned buffer. In that case, the *error* value of 0D indicates that communication with the I/O subsystem was successful, while the I/O subsystem logical status value reflects the status of the I/O subsystem.
- Searching logical devices

  To perform a search of logical devices, call the file-name inquiry procedures FILENAME_FINDSTART_, FILENAME_FINDNEXT_, and FILENAME_FINDFINISH_.

CONTIME Procedure

Summary

The CONTIME procedure converts a 48-bit timestamp to a date and time in integer form.

Syntax for C Programmers

```
#include <cextdecs(CONTIME)>

void CONTIME ( short _near *date-and-time
               ,short t0
               ,short t1
               ,short t2 );
```

Syntax for TAL Programmers

```
CALL CONTIME ( date-and-time            ! o
               ,t0                      ! i
               ,t1                      ! i
               ,t2 );                   ! i
```

Parameters

date-and-time output

INT:ref:7

is an array where CONTIME returns a date and time in this form:

[0]	year	(1975, 1976, ...)
[1]	month	(1-12)
[2]	day	(1-31)
[3]	hour	(0-23)
[4]	minute	(0-59)
[5]	second	(0-59)
[6]	0.01 second	(0-99)

t0, t1, t2 input

INT:value:3

is an array that must correspond to the 48 bits of a timestamp for the results of CONTIME to have any meaning (*t0* is the most significant word; *t2* is the least significant).

t0	most significant word, interval clock
t1	interval clock
t2	least significant word, interval clock

Considerations

- A 48-bit timestamp is a quantity equal to the number of 10 millisecond units since 00:00, 31 December 1974. The 48-bit timestamp always represents local civil time.
- Procedures that work with the 48-bit timestamp are CONTIME, TIME, and TIMESTAMP.
- For a more complete description of 48-bit and 64-bit timestamps, see TIMESTAMP Procedure or JULIANTIMESTAMP Procedure.

Example

```
CALL CONTIME( DATE^TIME , LAST^T , LAST^T[1] , LAST^T[2] );
          !conversion to date and time
```

CONTIME is used to convert the three words in LAST^T to a date and time. DATE^TIME returns seven words of date and time.

Related Programming Manual

For programming information about the CONTIME utility procedure, see the *Guardian Programmer's Guide*.

CONTROL Procedure

Summary

CONTROL is used to perform device-dependent I/O operations.

Syntax for C Programmers

```
#include <cextdecs(CONTROL)>

_cc_status CONTROL ( short filenum
                   ,short operation
                   ,[ short param ]
                   ,[ __int32_t tag ] );
```

- CEXTDECS (through the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.
- The function value returned by CONTROL, which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL CONTROL ( filenum                          ! i
             ,operation                         ! i
             ,[ param ]                         ! i
             ,[ tag ] );                        ! i
```

Parameters

filenum input

INT:value

is a number of an open file, identifying the file to which the CONTROL procedure performs an I/O operation.

operation input

INT:value

is a value that defines an operation to be performed (see Table 3-4 on page 3-109 and Table 3-5 on page 3-113 for a list of operation numbers).

param input

INT:value

is the value of an operation being performed (see Table 3-4 on page 3-109 and Table 3-5 on page 3-113).

tag input

INT(32):value

applies to nowait I/O only. The *tag* parameter is a value you define uniquely identifying the operation associated with this CONTROL call.

Note. The system stores the *tag* value until the I/O operation completes. It then returns the *tag* information to the program in the *tag* parameter of the call to AWAITIO, thus indicating that the operation finished.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILE_GETINFO_ or FILEINFO).

= (CCE) indicates that the CONTROL was successful.

> (CCG) for magnetic tape, indicates that the end of file (EOF) was encountered while spacing records; for a process file, this setting indicates that the process is not accepting process CONTROL messages. When device handlers do not allow the operation, file-system error 2 returns.

Control Operations

Table 3-4 and Table 3-5 on page 3-113 list the CONTROL operations that can be used with the I/O devices discussed in this manual for NonStop servers. Table 3-4 describes CONTROL operation 1 only; Table 3-5 on page 3-113 describes the remaining CONTROL operations.

Table 3-4. CONTROL Operation 1 (page 1 of 4)

Description	Subtype	Description of <param>	
Terminal or Line Printer Forms Control	0,2, or 3	0	form feed (send %014)
		1–15	vertical tab (send %013)
		16–79	skip <param> - 16 lines
Line Printer	6 or 32	0	form feed (send %014)
		1–15	single space (send %6412)
		16–79	skip <param> - 16 lines

Table 3-4. CONTROL Operation 1 (page 2 of 4)

Description	Subtype	Description of <param>	
Line Printer	1 or 5	0	skip to VFU channel 0 (top of form)
		1	skip to VFU channel 1 (bottom of form)
		2	skip to VFU channel 2 (single space, top-of form eject)
		3	skip to VFU channel 3 (next odd-numbered line)
		4	skip to VFU channel 5 (next one-half page)
		5	skip to VFU channel 5 (next one-half page)
		6	skip to VFU channel 6 (next one-fourth page)
		7	skip to VFU channel 7 (next one-sixth page)
		8	skip to VFU channel 8 (user-defined)
		9	skip to VFU channel 9 (user-defined)
		10	skip to VFU channel 10 (user-defined)
		11	skip to VFU channel 11 (user-defined)
		16–31	skip <param> - 16 lines

Table 3-4. CONTROL Operation 1 (page 3 of 4)

Description	Subtype	Description of <param>	
Line Printer	7	0	select VFC channel 1 (top of form)
		1	select VFC channel 2 (bottom of form)
		2	select VFC channel 3 (single space)
		3	select VFC channel 4 (skip to next double space line)
		4	select VFC channel 5 (skip to next triple space line)
		5	select VFC channel 6 (skip to next half page)
		6	select VFC channel 7 (skip to next one-fourth page)
		7	select VFC channel 8 (skip to next one-sixth page)
		8	select VFC channel 9 (bottom of form)
		9	select VFC channel 10 (bottom of form - 1)
		10	select VFC channel 11 (top of form - 1)
		11	select VFC channel 12 (top of form)
		16–31	skip <param> - 16 lines

Table 3-4. CONTROL Operation 1 (page 4 of 4)

Description	Subtype	Description of <param>	
Line Printer (default DAVFU)	4	0	skip to VFU channel 0 (top of form/line 1)
		1	skip to VFU channel 1 (bottom of form/line 60)
		2	skip to VFU channel 2 (single space/line 1-60, top-of-form eject)
		3	skip to VFU channel 3 (next odd-numbered line)
		4	skip to VFU channel 6 (next third line: 1,4,7,10, and so forth)
		5	skip to VFU channel 5 (next one-half page)
		6	skip to VFU channel 6 (next one-fourth page)
		7	skip to VFU channel 7 (next one-sixth page)
		8	skip to VFU channel 8 (line 1)
		9	skip to VFU channel 9 (line 1)
		10	skip to VFU channel 10 (line 1)
		11	skip to VFU channel 11(bottom of paper/line 63)
		16–31	skip <param> - 16 lines

Table 3-5. CONTROL Operations 2 Through 27 (page 1 of 3)

Operation	Description	Description of <param>
2	Write end of file on unstructured disk or magnetic tape (if disk, write access is required). A write end of file (EOF) to an unstructured disk file sets EOF to point to the relative byte address indicated by the next-record pointer and writes the new EOF setting in the file label on disk. If this new EOF setting is out of bounds, EOF is set to the last possible position. Note: CONTROL 2 is valid only for unstructured files or structured files opened for unstructured access. For XP-based files, using CONTROL 2 to move the EOF will involve more CPU cycles. Writing the application data to the file and letting the disk process move the EOF as the data is written to the file, and removing the CONTROL 2 operations from the application code will improve the performance.	None
3	Magnetic tape, rewind and unload, do not wait for completion.	None
4	Magnetic tape, take off line, do not wait for completion (treated as operation 3 for 5106 Tri-Density Tape Drive). Not supported for 5130, 5160, 5170, and 5180 tape drives.	None
5	Magnetic tape, rewind leave on line, do not wait for completion.	None
6	Magnetic tape, rewind, leave on line, wait for completion.	None
7	Magnetic tape, space forward files.	number of files {0:255}
8	Magnetic tape, space backward files.	number of files {0:255}
9	Magnetic tape, space forward records.	number of records {0:255}
10	Magnetic tape, space backward records.	number of records {0:255}
11	Terminal or serial-connected line printer, wait for modem connect.	None
12	Terminal or serial-connected line printer, disconnect the modem (that is, hang up).	None

Table 3-5. CONTROL Operations 2 Through 27 (page 2 of 3)

Operation	Description	Description of <param>
13	Issue an SNA CHASE request. SNAX exception response mode must be enabled to use operation 13.	None
17	Terminal or serial-connected line printer, enable connection and initiate call to remote DTE in X.25 network.	None
20	Disk, purge data (write access is required).	None
21	Disk, allocate or deallocate extents (write access is required).	0 deallocate all extents past the end-of-file extent 1:*maximum-extents* number of extents to allocate for a nonpartitional file (for DP2 disk files only) 1:16*number of partitions number of extents to allocate for a partitioned file
22	Cancel an AM3270 I/O operation.	None
24	Magnetic tape, force end-of-volume (EOV). Next volume in set is requested and current volume is unloaded. Valid only for ANSI or IBM label tape.	None

Table 3-5. CONTROL Operations 2 Through 27 (page 3 of 3)

Operation	Description	Description of <param>
26	Requests immediate completion of all outstanding I/O requests without loss of data by the recipient of the CONTROL 26 request.	None
27	Wait for DP2 disk file write. This operation is not supported for queue files. This operation finishes when a WRITE, WRITEUPDATE, or WRITEUPDATEUNLOCK occurs on a DP2 disk file designated by *filenum*. Not valid for partitioned files. Do not assume that the file contains any new data when a call to CONTROL 27 finishes; assume only that it is time to check the file for new data. CONTROL 27 also finishes when a WRITE or WRITEUPDATE occurs as part of a logical undo of a transaction by the backout process or when a volume goes down. If SETMODE 146 was specified, each write completes only one CONTROL 27; otherwise each write completes all pending CONTROL 27 operations. To ensure that no updates are missed, you should issue a nowait CONTROL 27 call on one open to a file, then read data from the file on another open, and finally wait for the CONTROL operation to finish. If CONTROL 27 were executed after reading, a write by another process could occur between the read and the CONTROL operations. The file open that is used for the CONTROL operation should have a *syncdepth* of 0. Path errors and network errors might indicate successful completion. See the discussion of CONTROL operation 27 in the *Guardian Programmer's Guide*.	

Considerations

- Nowait and CONTROL

  If the CONTROL procedure is used on a file that is opened nowait, it must be completed with a call to the AWAITIO procedure.

- Disk files

  - Writing EOF to an unstructured file

    Writing EOF to an unstructured disk file sets the EOF pointer to the relative byte address indicated by the setting of the next-record pointer and writes the new EOF setting in the file label on disk. Specifically, write:

    ```
    end-of-file pointer := next-record pointer;
    ```

    (File pointer action for CONTROL operation 2, write EOF.)

    Note: CONTROL 2 is valid only for unstructured files or structured files opened for unstructured access. For XP-based files, using CONTROL 2 to move the EOF will involve more CPU cycles. Writing the application data to the file and letting the disk process move the EOF as the data is written to the file, and removing the CONTROL 2 operations from the application code will improve the performance.

  - File is locked

    If a CONTROL operation is attempted for a file locked through a *filenum* other than that specified in the call to CONTROL, the call is rejected with a “file is locked” error 73.

    If any record is locked in a file, a call to CONTROL to write EOF (operation 2) to that same file will be rejected with a “file is locked” error 73.

- Magnetic tapes

  - When device is not ready

    If a magnetic tape rewind is performed concurrently with application program execution (that is, rewind operation <> 6), any attempt to perform a read, write, or control operation to the rewinding tape unit while rewind is taking place results in an error indication. A subsequent call to FILE_GETINFO_ or FILEINFO shows that an error 100 occurred.

  - Wait for rewind to complete

    If a magnetic tape rewind operation = 6 (wait for completion) is performed as a nowait operation, the application waits at the call to AWAITIO for the rewind to complete.

- Interprocess communication
  - Nonstandard *operation* and *parameter* values

    Any value can be specified for the *operation* and *parameter* parameters. An application-defined protocol should be established for interpreting nonstandard parameter values.
  - Process not accepting system messages

    If the object of the control operation is not accepting process CONTROL messages, the call to CONTROL completes with a condition code of CCG; a subsequent call to FILE_GETINFO_ or FILEINFO shows that an error 7 occurred.
  - Process control

    You can obtain the process identifier of the caller to CONTROL in a subsequent call to FILE_GETRECEIVEINFO_ (or LASTRECEIVE or RECEIVEINFO).

Related Programming Manuals

For programming information about the CONTROL file-system procedure, see the *Guardian Programmer's Guide*, the *Enscribe Programmer's Guide*, and the data communications manuals.

CONTROLBUF Procedure

Summary

The CONTROLBUF procedure is used to perform device-dependent I/O operations requiring a data buffer.

Syntax for C Programmers

```
#include <cextdecs(CONTROLBUF)>

_cc_status CONTROLBUF ( short filenum
                      ,short operation
                      ,short _near *buffer
                      ,short count
                      ,[ short _near *count-transferred ]
                      ,[ __int32_t tag ] );
```

- CEXTDECS (through the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.
- The function value returned by CONTROLBUF, which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL CONTROLBUF ( filenum                          ! i
                 ,operation                        ! i
                 ,buffer                           ! i
                 ,count                            ! i
                 ,[ count-transferred ]            ! o
                 ,[ tag ] );                       ! i
```

Parameters

filenum input

INT:value

is the number of an open file. It identifies the file on which the CONTROLBUF procedure performs an I/O operation.

operation input

INT:value

is a value defined by the device, such as:

`1` = Load DAVFU (printer subtype 4)

buffer input

INT:ref:*

is an array that contains the information to be used for the CONTROLBUF operation.

count input

INT:value

is the number of bytes contained in *buffer*.

count-transferred output

INT:ref:1

returns a count of the number of bytes transferred from *buffer* (for wait I/O only).

tag input

INT(32):value

is for nowait I/O only. The *tag* parameter is a value you define that uniquely identifies the operation associated with this CONTROLBUF call.

Note. The system stores the *tag* value until the I/O operation completes. The system then returns the *tag* information in the *tag* parameter of the call to AWAITIO, thus indicating that the operation finished.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILE_GETINFO_ or FILEINFO).

= (CCE) indicates that the CONTROLBUF was successful.

> (CCG) for a process file, indicates that the process is not accepting process CONTROLBUF messages.

Considerations

- Wait and *count-transferred*

  If a waited CONTROLBUF is executed, the *count-transferred* parameter indicates the number of bytes actually transferred.

- Nowait and *count-transferred*

  If a nowait CONTROLBUF is executed, *count-transferred* has no meaning and can be omitted. A count of the number of bytes transferred is obtained by the *count-transferred* parameter of the AWAITIO procedure when the I/O finishes.

  The CONTROLBUF procedure must complete with a call to the AWAITIO procedure when used with a file opened nowait.

- When object of CONTROLBUF is not accepting messages

  If the object of the CONTROLBUF operation is not accepting process CONTROLBUF messages, the call to CONTROLBUF completes with condition code CCG. A subsequent call to FILE_GETINFO_ or FILEINFO shows that an

error 7 (process not accepting process CONTROL, CONTROLBUF, or SETMODE messages) occurred.

You can obtain the process identifier of the caller to CONTROLBUF in a call to FILE_GETRECEIVEINFO_ (or LASTRECEIVE or RECEIVEINFO) after you have read the process CONTROLBUF message.

- Nonstandard *operation* and *buffer* parameters

  You can specify any value for the *operation* parameter, and you can include any data in *buffer*. An application-defined protocol should be established for interpreting nonstandard parameter values.

Messages

- Process CONTROLBUF message

  Issuing a CONTROLBUF to a file that represents another process causes a system message -35 (process CONTROLBUF) to be sent to that process. For detailed information of system messages sent to processes, see the *Guardian Procedure Errors and Messages Manual*.

CONTROLMESSAGESYSTEM Procedure

Summary

The CONTROLMESSAGESYSTEM procedure controls the maximum number of receive and send messages used by a process.

Syntax for C Programmers

```
#include <cextdecs(CONTROLMESSAGESYSTEM)>

short CONTROLMESSAGESYSTEM ( short actioncode
                            ,short value );
```

Syntax for TAL Programmers

```
error := CONTROLMESSAGESYSTEM ( actioncode          ! i
                               ,value );            ! i
```

Parameters

error returned value

INT:ref:1

returns error number:

0	Successful, no error
2	Bad *actioncode*
21	Bad *value*
29	Missing parameter

actioncode input

INT:value

specifies the action to be taken. See this list of action codes and values.

value input

INT:value

supplies a value to be used in taking an action. See this list of action codes and values.

action code	value	Action Taken
0	1 through 4095 * 1 through 16383 **	This value (considered an unsigned number) sets the limit on the number of outstanding incoming messages to the process. The default limit is 255; opening $RECEIVE with receive depth can increase this limit. The elapsed-time timeout messages and system status messages are not affected by this limit.
1	1 through 4095 * 1 through 16383 **	This value (considered an unsigned number) sets the limit on the number of outstanding messages sent by the process. The default limit is 1023. The elapsed-time timeout messages and system status messages are not affected by this limit.
2	N/A	Beginning in the D-series RVU, this action code of the operating system is obsolete.
3	N/A	Beginning in the D-series RVU, this action code of the operating system is obsolete.

* For systems running G06.29 and later G-series RVUs
** For systems running H06.06 and later H-series RVUs

Considerations

- If a requester tries to send a message to a process that already has as many incoming messages as allowed (as specified by action code 0), then the message is not sent, and the requester receives an error 30. If a time limit set by a process expires (such as a time limit set with SIGNALTIMEOUT), then a message notifying the process of the expiration is sent to the process anyway, and is not counted as part of that limit.
- If a process that already has as many outstanding outgoing messages as allowed (as specified by action code 1) tries to send a message, the process receives an error 30.
- CONTROLMESSAGESYSTEM is tied to the internal operation of the message system—current functions might not be supported by future versions of the message system. So, if a nonzero *error* is returned, the caller should log the error, but otherwise ignore it.
- For information on measuring process message requirements, see the MESSAGESYSTEMINFO Procedure.

- Before D-series RVUs of the operating system, the RESERVELCBS procedure allowed you to reserve message-system control blocks. It also performed certain secondary functions concerning limits. Beginning in the D-series RVU of the operating system, reserving message-system control blocks is no longer applicable, and the secondary functions of the procedure are accomplished by calling CONTROLMESSAGESYSTEM. Although RESERVELCBS no longer performs any function, it can still be called without error.

CONVERTASCIIEBCDIC Procedure

Summary

This procedure translates the 256 EBCDIC encodings to and from the 256 8-bit ASCII encodings. For more information, see Appendix K, Character Set Translation.

Syntax for C Programmers

```
#include <cextdecs(CONVERTASCIIEBCDIC)>

void CONVERTASCIIEBCDIC ( const char* buffer
                        ,const unsigned short count
                        ,const short translation);
```

Syntax for TAL Programmers

```
CALL CONVERTASCIIEBCDIC ( buffer                    !i
                        ,count                      ! i
                        ,translation);               ! i
```

Parameters

buffer input

STRING .EXT:ref:*

points to the start of an array of characters to be translated.

count input

INT:value

specifies the number of characters to convert, in the range 0 through 65535.

translation input

INT:value

specifies these translations:

0: None
1: EBCDIC to ASCII
2: ASCII to EBCDIC

All other values have undefined effects.

CONVERTPROCESSNAME Procedure (Superseded by FILENAME_RESOLVE_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The CONVERTPROCESSNAME procedure converts a process name from local to network form.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
CALL CONVERTPROCESSNAME ( process-name );          ! i,o
```

Parameters

process-name input, output

INT:ref:3

is the process name beginning with “$” to be converted.

On return, *process-name* contains the internal network form of the process name: "\" in the first byte and the calling process's system number in the second byte, followed by the process name.

If *process-name* does not begin with "$", it is left unchanged.

Considerations

CONVERTPROCESSNAME truncates any process name that is longer than four characters plus the "$".

CONVERTPROCESSTIME Procedure

Summary

The CONVERTPROCESSTIME procedure is used to convert the quad microsecond process time returned by the PROCESSTIME, MYPROCESSTIME, or PROCESSINFO procedure into hours, minutes, seconds, milliseconds, and microseconds. The maximum time that this procedure can convert is 3.7 years (the amount of time that can be represented using the output parameters).

Syntax for C Programmers

```
#include <cextdecs(CONVERTPROCESSTIME)>

_cc_status CONVERTPROCESSTIME ( long long process-time
                              ,[ short _near *hours ]
                              ,[ short _near *minutes ]
                              ,[ short _near *seconds ]
                              ,[ short _near *milliseconds ]
                              ,[ short _near *microseconds ] );
```

- The function value returned by CONVERTPROCESSTIME, which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL CONVERTPROCESSTIME ( process-time                ! i
                        ,[ hours ]                    ! o
                        ,[ minutes ]                  ! o
                        ,[ seconds ]                  ! o
                        ,[ milliseconds ]             ! o
                        ,[ microseconds ] );          ! o
```

Parameters

process-time input

FIXED:value

specifies the time to be converted.

hours output

INT:ref:1

returns the hours portion of the value of *process-time* specified.

minutes output

INT:ref:1

is the minutes portion of the value of *process-time* specified.

seconds output

INT:ref:1

is the seconds portion of the value of *process-time* specified.

milliseconds output

INT:ref:1

is the milliseconds portion of the value of *process-time* specified.

microseconds output

INT:ref:1

is the microseconds portion of the value of *process-time* specified.

Condition Code Settings

< (CCL) returns if *process-time* represents a quantity greater than 3.7 years.

= (CCE) indicates that CONVERTPROCESSTIME is successful.

> (CCG) returns if any of the supplied output parameters fails the bounds check on the address.

Related Programming Manual

For programming information about the CONVERTPROCESSTIME procedure, see the *Guardian Programmer's Guide*.

CONVERTTIMESTAMP Procedure

Summary

The CONVERTTIMESTAMP procedure converts a Greenwich mean time (GMT) timestamp to or from a local-time-based timestamp within any accessible node in the network.

A local timestamp can be in local standard time (LST), which does not include daylight saving time (DST), or in local civil time (LCT), which does include DST.

DST is a system to extend the amount of daylight hours available in summer by putting the clock forward by an hour. Before 2007, DST in the United States, begins at 2:00 a.m. on the first Sunday of April and ends at 2:00 a.m. DST (1:00 a.m. LST) on the last Sunday of October. From 2007 onwards, DST will begin at 2:00 a.m. on the second Sunday of March and end at 2:00 a.m. DST (1:00 a.m. LST) on the first Sunday of November.

Syntax for C Programmers

```
#include <cextdecs(CONVERTTIMESTAMP)>

long long CONVERTTIMESTAMP( long long julian-timestamp
                          ,[ short direction ]
                          ,[ short node ]
                          ,[ short _near *error ] );
```

Syntax for TAL Programmers

```
ret-time := CONVERTTIMESTAMP ( julian-timestamp       ! i
                             ,[ direction ]           ! i
                             ,[ node ]                ! i
                             ,[ error ] );            ! o
```

Parameters

ret-time returned value

FIXED

returns the converted Julian timestamp.

julian-timestamp input

FIXED:value

is a four-word Julian timestamp to be converted.

direction input

INT:value

indicates what time form or timestamp to return. You can specify one of these values for *direction*.

0 GMT to LCT
1 GMT to LST
2 LCT to GMT
3 LST to GMT

If *direction* is omitted, 0 is used. If *direction* is out of range, a value of -3 is returned in *error*.

node input

INT:value

is the number of the node at which the conversion is to take place. If the *node* parameter is omitted or -1, the caller's node is used. If the specified node does not exist, the value of *ret-time* is not changed.

error output

INT:ref:1

returns one of these values:

-5	Value of *node* is out of range
-4	Timestamp not supplied or has invalid value
-3	Invalid value supplied for *direction*
-2	Impossible LCT
-1	Ambiguous LCT
0	No errors, successful
1	DST range error
2	DST table not loaded
>2	File-system error (attempting to reach *node*)

Considerations

- A local timestamp can be in either of two forms: LCT (with DST correction) or LST (without DST correction).
- Network and local timestamp

  Local timestamp (with LCT and LST) should be used with caution if any network use is anticipated. The reason is that another node can be in another time zone or in an area with different DST rules (LCT only).
- LCT timestamps

  LCT timestamps should be used with caution because of the negative adjustment that DST systems dictate. Timestamp base conversion (for example, LCT) is provided by the operating system.
- A 64-bit Julian timestamp is based on the Julian Date. It is a quantity equal to the number of microseconds since January 1, 4713 B.C., 12:00 (noon) Greenwich mean time (Julian proleptic calendar). This timestamp can represent either Greenwich mean time, local standard time, or local civil time. There is no way to examine a Julian timestamp and determine which of the three times it represents.

  Procedures that work with the 64-bit Julian timestamp are COMPUTETIMESTAMP, CONVERTTIMESTAMP, INTERPRETTIMESTAMP, JULIANTIMESTAMP, and SETSYSTEMCLOCK.

For a more complete description of 48-bit and 64-bit timestamps, see TIMESTAMP Procedure or JULIANTIMESTAMP Procedure.

- Before 1987, DST in the United States started on the last Sunday of April. Before 2007, DST in the United States begins on the first Sunday of April and ends on the last Sunday of October. From 2007 onwards, DST will begin on the second Sunday of March and will end on the first Sunday of November.
- Timestamps occurring before 1987 and 2007 are correctly converted in accordance with the previous standards.
- Setting up a DST table

  If your system is configured to use a DST table, add entries carefully to the DST table to avoid problems in the conversion of timestamps by CONVERTTIMESTAMP.

  These problems can arise when your system uses a DST table:

  - Inaccurate conversions because of wrong information in the DST table
  - Programs abending or taking other extreme actions in response to errors reported by CONVERTTIMESTAMP

  It is important to set up the DST table to prevent CONVERTTIMESTAMP reporting error 1 (DST range error) or error 2 (DST table not loaded) because many programs take extreme actions in response to these errors. For example, BIND is known to abend in response to error 1 or error 2, and SQLCOMP fails in response to BIND abending. BACKUP and RESTORE have also been known to fail in response to error 1 or error 2.

  To allow CONVERTTIMESTAMP to perform accurate conversions from GMT to LCT, it is important to provide accurate information for DST transitions for all timestamps that CONVERTTIMESTAMP is likely to encounter. The accurate DST table entries should go back at least several years, to handle such things as timestamps for files that have been created in the past. You should also provide the most accurate DST transition information available for several years into the future. DST transition information should go at least one year farther back and one year farther in the future than you expect to encounter timestamps.

  It is vital to add at least one period of nonzero DST offset to the DST table, to avoid CONVERTTIMESTAMP reporting error 2 (DST table not loaded).

  In addition to the accurate table entries recommended above, it is good practice to add fictitious entries to the DST table, to avoid CONVERTTIMESTAMP reporting error 1 (DST range error) when it converts times earlier than expected or later than expected. It is good practice to add a fictitious entry for a time far in the past and a fictitious or speculative entry for a time far in the future. For example, you might add DST table entries for the year 1000 and for the year 3999.

To add entries to the DST table, use these procedures:

D-series RVUs, or G04.00 and earlier G-series RVUs	ADDDSTTRANSITION procedure or the ADDDSTTRANSITION TACL command
G05.00 and later G-series RVUs	ADDDSTTRANSITION procedure, ADDDSTTRANSITION TACL command, or the DST_TRANSITION_ADD_ procedure

- Use (or Avoidance) of Local Civil Time

  It is hard to avoid all problems with conversion between local civil time (which includes the effect of Daylight Saving Time) and GMT or local standard time.

  Because civil time is convenient for people to use for comparison with local clocks, some use of civil time is expected. Accordingly, the best strategy might be to store all timestamps in GMT, and then to convert the GMT timestamps to civil time, when desired, before displaying them. It might also be helpful to display the GMT timestamp. This would allow people to clarify cases such as timestamps near a DST transition or timestamps that might have been generated on a different system, possibly in a different time zone.

  If a program restricts itself to converting from GMT to LCT at the local system, and never converts in the opposite direction or at a remote node, then the errors that might unexpectedly occur are error 1 (DST range error) or 2 (DST table not loaded). In both of these cases, the conversion from GMT to LCT will assume a DST offset of 0. If Daylight Saving Time is not in effect for the time in question then the timestamp will be correct. Otherwise, it will be incorrect by the DST offset, which is typically one hour.

  When converting from GMT to LCT at the local system, it is best to use the value returned from CONVERTTIMESTAMP without checking the error parameter. In rare cases, such as when comparing timestamps to decide whether to rebuild a file, it might be better to take some special action as a safety precaution, such as adding or subtracting one hour from the timestamp.

- Error Handling

  Whenever a program uses CONVERTTIMESTAMP to convert to or from civil time, there is a possibility that CONVERTTIMESTAMP will report a nonzero value of error. It is very undesirable for the program to take any extreme action such as abending or failing a transaction because of a such a conversion error.

  It is particularly important to avoid abending or other extreme actions when responding to errors in time conversions that might be several years in the past or

the future. It is difficult for operators of a computer system to provide accurate information about future and past Daylight Saving Time data.

Example

```
#include  <cextdecs (CONVERTTIMESTAMP)>
#include <ktdmtyp.h>  /* define long long etc. */

long long gmt_time;  /* original (GMT) time stamp */
long long display_time;   /* displayable (Local Civil Time) time stamp */

display_time = CONVERTTIMESTAMP( gmt_time, 0 ); /* GMT to LCT */
```

Related Programming Manual

For programming information about the CONVERTTIMESTAMP procedure, see the *Guardian Programmer's Guide*.

CPU_GETINFOLIST_ Procedure

Use the PROCESSOR_GETINFOLIST_ Procedure instead of CPU_GETINFOLIST_. Calls to PROCESSOR_GETINFOLIST_ are identical in their format and values to those for CPU_GETINFOLIST_.

CPUTIMES Procedure

Summary

The CPUTIMES procedure returns the length of time, in microseconds, that a given processor has spent in these states since it was loaded or reloaded:

- Process busy
- Interrupt busy
- Idle

These times reflect the amount of time spent by the processor (from the last system load or reload) in a process environment, an interrupt environment, and the idle state.

Syntax for C Programmers

```
#include <cextdecs(CPUTIMES)>

_cc_status CPUTIMES( [ short cpu ]
                    ,[ short sysid ]
                    ,[ long long *total-time ]
                    ,[ long long *cpu-process-busy ]
                    ,[ long long *cpu-interrupt ]
                    ,[ long long *cpu-idle ] );
```

- The function value returned by CPUTIMES, which indicates the condition code, can be interpreted by the `_status_lt()`, `_status_eq()`, or `_status_gt()` function (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL CPUTIMES ( [ cpu ]                          ! i
               ,[ sysid ]                        ! i
               ,[ total-time ]                   ! o
               ,[ cpu-process-busy ]             ! o
               ,[ cpu-interrupt ]                ! o
               ,[ cpu-idle ] );                  ! o
```

Parameters

cpu input

INT:value

specifies the processor number of a processor in the system. The default is the local processor.

sysid input

INT:value

specifies the system number. The default is the local system.

total-time output

FIXED:ref:1

returns the elapsed time, in microseconds as measured by the processor clock, since the processor was loaded or reloaded.

cpu-process-busy output

FIXED:ref:1

returns the length of time, in microseconds as measured by the processor clock, that the processor has been busy executing processes since it was loaded or reloaded.

cpu-interrupt output

FIXED:ref:1

returns the length of time, in microseconds as measured by the processor clock, that the processor has been busy processing interrupts since it was loaded or reloaded.

cpu-idle output

FIXED:ref:1

returns the length of time, in microseconds as measured by the processor clock, that the processor has been idle since it was loaded or reloaded.

Condition Code Settings

< (CCL) indicates that the system is in one of these states:

- Unavailable.
- Does not exist.
- The procedure could not get resources to execute.

= (CCE) indicates that CPUTIMES is successful.

> (CCG) indicates that supplied parameters failed the bounds check.

CREATE Procedure (Superseded by FILE_CREATELIST_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The CREATE procedure is used to define a new structured or unstructured disk file. The file can be temporary (and therefore automatically deleted when closed) or permanent. When a temporary file is created, CREATE returns its file name in a form suitable for passing to the OPEN procedure.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
CALL CREATE ( file-name                          ! i,o
            ,[ primary-extentsize ]              ! i
            ,[ file-code ]                       ! i
            ,[ secondary-extentsize ]            ! i
            ,[ file-type ]                       ! i
            ,[ recordlen ]                       ! i
            ,[ data-blocklen ]                   ! i
            ,[ key-sequenced-params ]            ! i
            ,[ alternate-key-params ]            ! i
            ,[ partition-params ]                ! i
            ,[ maximum-extents ]                 ! i
            ,[ unstructured-buffer-size ]        ! i
            ,[ open-defaults ] );                ! i
```

Parameters

file-name input, output

INT:ref:12

is an array containing the internal-format file name of the disk file to be created. The value of *file-name,* must be in one of these forms (to create a permanent or temporary disk file):

Permanent Disk File

[0:3] $*volname* (blank-fill)

or

sysnum volname (blank-fill)

[4:7] *subvol-name* (blank-fill)

[8:11] *file-id* (blank-fill)

Temporary Disk File

[0:3] $*volname* (blank-fill)

or

sysnum volname (blank-fill)

[4:11] blank-fill

When CREATE finishes, a temporary file name is returned in *file-name* [4:7]. The temporary file can then be opened by passing *file-name* to OPEN.

primary-extentsize input

INT:value

is the size of the primary extent in pages (one page is 2048 bytes). The maximum value of *primary-extentsize* is 65,535 (134,215,680 bytes). If omitted, a primary extent size of 1 page is assigned.

file-code input

INT:value

is an application-defined file identification code (file codes 100-999 are reserved for use by HP). If omitted, a file code of 0 is assigned. For a list of HP file codes, see the *Guardian Utilities Reference Manual*.

secondary-extentsize input

INT:value

is the size of the secondary extents in pages (one page is 2048 bytes). (The maximum number of secondary extents that a file can have allocated is *maximum-extents* - 1. See *maximum-extents*, below.) The maximum value of *secondary-extentsize* is 65535 (134,215,680 bytes). If omitted, the size of the primary extent is used for the secondary extent size.

file-type input

INT:value

specifies the type of file to be created. If omitted, an unstructured file is created.

<0:1> Must be 0.

<2> In systems with the Transaction Management Facility (TMF), specifies that this file is audited; for systems without TMF, this bit is 0.

<3:8> Must be 0.

<9> Specifies that this file is a queue file.

<10> specifies that the file label is written to disk each time the end of file (EOF) is advanced.

<11> Specifies index compression for key-sequenced files (see the *Enscribe Programmer's Guide*).

<12> Specifies ODDUNSTR access to unstructured files. With the default (*file-type*.<12> = 0), a relative byte address (RBA) used for reading, writing, or positioning in the file, is rounded up to the next even number (whole word boundary); thus, 3 rounds up to 4, and so forth. ODDUNSTR prevents this rounding, so that reading, writing, or positioning occurs at the exact RBA specified. (See "Considerations.")

<12> Specifies data compression for key-sequenced files. (For additional information, see the *Enscribe Programmer's Guide*.)

<13:15> Specifies the file structure:

0 Unstructured (default)
1 Relative
2 Entry-sequenced
3 Key-sequenced

recordlen input

INT:value

is the maximum length of the logical record in bytes. If omitted, 80 is used. This parameter is ignored for unstructured files.

The formulas for computing the maximum record size (MRS) based on *data-blocklen* are:

For this type of file	MRS equals
Relative and entry-sequenced files	*data-blocklen* - 24
Key-sequenced files	*data-blocklen* - 34
Unstructured files	4096

data-blocklen input

INT:value

for structured files, is the length in bytes of each block of records in the file. The value of *data-blocklen* cannot be greater than 4096. The value of *data-blocklen* must be at least *recordlen* + 24. For a key-sequenced file, the value of *data-blocklen* must be at least *recordlen* + 34.

If omitted, 1024 is the default value used for *data-blocklen*. Regardless of the specified record length and data-block size, the maximum number of records that can be stored in a data block is 511.

Data-block sizes are rounded up to power-of-two multiples of the sector size; 512, 1024, 2048, and 4096. For example, if a 3K byte block were specified, the system would use 4096.

key-sequenced-params input

INT:ref:3

is a three-word array containing parameters that describe this file. This parameter is required for key-sequenced files, but you can omit the parameter for other file types. See “Considerations” for the format of this array.

alternate-key-params input

INT:ref:*

is an array containing parameters describing any alternate keys for this file. This parameter is required if the file has alternate keys; otherwise, you can omit this parameter. If included, the first word must be 0 if you do not have alternate keys. See “Considerations” for the format of this array.

partition-params input

INT:ref:*

is an array containing parameters that describe this file. It applies only if the file is a multivolume file. If the file is to span multiple volumes, this parameter is required; otherwise, you can omit it. If included, and you do not want partitions, the first word must be 0. See “Considerations” for the format of this array.

maximum-extents input

INT:value

is the maximum number of extents to be allocated for the file. The minimum and default value is 16. See “Considerations,” for the upper limit on this value.

unstructured-buffer-size input

INT:value

declares the internal buffer size to be used for an unstructured file. Must be 512, 1024, 2048, or 4096. The default is 4096 bytes.

open-defaults input

INT:value

specifies the file label default values for various open attributes.

<0>	0	Verify writes off (default)
	1	Verify write on
<1>	0	System automatically selects serial or parallel writes

	1	Serial mirror writes only
<2>	0	Buffered writes enabled (default for audited files)
	1	Write-thru (default for nonaudited files)
<3>	0	Audit compression off (default)
	1	Audit compression on

Condition Code Settings

< (CCL) indicates that the CREATE failed (call FILEINFO or FILE_GETINFO_).

= (CCE) indicates that the file was created successfully.

> (CCG) indicates that the device is not a disk.

Considerations

- *key-sequenced-params* array format

Word[0]	*key-len*
Word[1]	*key-offset*
Word[2]	*index-block-len*

key-len

(INT:value)

is the length, in bytes, of the record's primary-key field. This length can be no larger than 255 bytes.

key-offset

(INT:value)

is the number of bytes from the beginning of the record to where the primary-key field starts. This attribute applies only to Enscribe files.

index-block-len

(INT:value)

was the length, in bytes, of each index block in the file on older systems. On current systems, the value of *data-blocklen* is used as the value of *index-block-len*.

- *alternate-key-params* array format

	0	8
Word[0]	*nf-alt-files*	*nk-alt-keys*
Word[1]	Key Description for Alternate Key 0	

	Key Description for Alternate Key nk - 1
[k * 4 + 1]	File Name of Key File 0
	File Name of Key File nf - 1

nf-alt-files

a 1-byte value, specifies the number of alternate-key files for this primary file.

nk-alt-keys

a 1-byte value, specifies the number of alternate-key fields in this primary file.

The key description for key k consists of four words, each of the form:

	0	8
[k * 4 + 1]	*key-specifier*	
[k * 4 + 2]	*key-attributes*	
[k * 4 + 3]	*null-value*	*key-len*
[k * 4 + 4]	*key-filenum*	

key-specifier

(INT:value)

is a 2-byte value that uniquely identifies this alternate-key field. It must be nonzero. This value is passed to the KEYPOSITION procedure for references to this key field.

key-attributes

(INT:value)

describes the key:

<0> 1 Means that a null value is specified. See *null-value*, below.

<1> 1 Means that the key is unique. If an attempt is made to insert a record that duplicates an existing value in this field, the insertion is rejected with a “duplicate record” error.

<2> 1 Means that Enscribe cannot perform automatic updating of this key.

<3> 0 Means that alternate key records with duplicate key values are ordered by the value of the primary record key field. This attribute has meaning only for alternate keys that allow duplicates.

1 Means that alternate key records with duplicate key values are ordered by the sequence in which those records were inserted into the

alternate key file. This attribute is allowed only for alternate keys that allow duplicates.

<4:15>*key-offset*
Specifies the number of bytes from the beginning of the record where this key field starts.

null-value

(BYTE:value)

a 1-byte value, is used to specify a null value if *key-attributes*.<0> is equal to 1.

During a write operation, if a null value is specified for an alternate-key field and if the null value is encountered in all bytes of this key field, the file system does not enter the reference to the record in the alternate-key file. (If the file is read using this alternate-key field, records containing a null value in this field will not be found.)

During a WRITEUPDATE operation (*write-count* = 0), if a null value is specified and if the null value is encountered in all bytes of this key field within *buffer*, the file system deletes the record from the primary file but does not delete the reference to the record in the alternate file.

key-len

(BYTE:value)

specifies the length, in bytes, of the alternate-key field. The maximum key length of an alternate key that allows duplicates and is defined as insertion-ordered (see *key-attributes*, above) is:

```
255 - (10 + primary key length)
```

key-filenum

(INT:value)

is the relative number in the alternate-key parameter array of this key's alternate-key file. The first alternate-key file's *key-filenum* = 0.

The file name for file f consists of 12 words, beginning at:

```
[nk * 4 + 1 + f * 12]
```

This file name has this form:

[0:3] $*volname* (blank-fill)
or
sysnum volname (blank-fill)

[4:7] *subvol-name* (blank-fill)

[8:11] *file-id* (blank-fill)

- *partition-params* array format

Number of Words		
	[1]	*num-of-extra-partitions*
	[4]	$*volname* or *sysnumvolname* for partition 1
		$*volname* or *sysnumvolname* for partition 2
		:
		$*volname* or *sysnumvolname* for partition n
	[1]	*primary-extent-size* part 1
		:
		primary-extent-size part n
	[1]	*secondary-extent-size* part 1
		secondary-extent-size part n

This sequence must be included in the partition-parameters array for key-sequenced files, but it can be omitted for other file types:

[1]	
	partial-keylen
	partial-keyvalue for partition 1
	:
	partial-keyvalue for partition n

num-of-extra-partitions

(INT:value)

is the number of extra volumes (other than the one specified in the *file-name* parameter) on which the file resides. The maximum value is 15. Note that every other parameter in the partition array (except *partial-keylen*) must be specified *num-of-extra-partitions* times.

$*volname* or *sysnumvolname*

8 bytes blank-filled, is the name of the disk volume (including the dollar sign ($) or backslash (\) where the particular partition is resides.

primary-extent-size

(INT:value)

is the size of the primary extent for the particular partition.

secondary-extent-size

(INT:value)

is the size of the secondary extents for the particular partition. Specifying 0 results in the *primary-extent-size* value being used.

The remaining parameters are required for key-sequenced files but can be omitted for all other file types:

partial-keylen

(INT:value)

is the number of bytes of the primary key of a key-sequenced file that are used to determine which partition of the file contains a particular record. The minimum value for *partial-keylen* is 1.

partial-keyvalue

(INT:value)

for *partial-keylen* bytes, specifies the lowest key value that is allowed for a particular partition.

Each *partial-keyvalue* in *partition-parameters* must begin on a word boundary.

For an alternate-key file, *partial-keyvalue* must begin with the *key-specifier* for the alternate key. For example, if *key-specifier* = AB, a partial-key value of 123 becomes a *partial-keyvalue* of AB123.

- File pointer action

  The end-of-file pointer is set to zero after the file is created.

- Disk allocation with CREATE

  Execution of the CREATE procedure does not allocate any disk area; it only provides an entry into the volume's directory, indicating that the file exists.

- CREATE failure

  If the CREATE fails (that is, condition code other than CCE returns), the reason for the failure can be determined by calling the FILEINFO or FILE_GETINFO_ procedure and passing -1 as the *filenum* parameter.

- Upper limit for *maximum-extents*

  There is no guarantee that a file will be created successfully if you specify a value greater than 500 for *maximum-extents*.

  In addition, CREATE returns error 21 if the values for *primary-extent-size*, *secondary-extent-size*, and *maximum-extents* yield a file size greater than (2**32) - 4096 bytes (approximately four gigabytes), or a partition size greater than 2**31 bytes (two gigabytes).

- Altering file security

  The file is created with the caller's process file security that can be examined and set with the PROCESSFILESECURITY procedure. Once a file has been created, its file security can be altered by opening the file and issuing the appropriate SETMODE and SETMODENOWAIT functions.

- Odd unstructured files

  An odd unstructured file permits reading and writing of odd byte counts and positioning to an odd byte address.

  When creating unstructured files, the value passed for *file-type*.<12> determines how all subsequent reading, writing, and positioning operations to the file work.

  If *file-type*.<12> is passed as 1 and *file-type*.<13:15> is all zeros, an odd unstructured file is created.

  If *file-type*.<12> is passed as 1, the values of *record-specifier*, *read-count*, and *write-count* are all interpreted exactly; for example, a *write-count* or *read-count* of 7 transfers exactly 7 bytes.

- Even unstructured files

  If *file-type*.<13:15> is passed to CREATE and is all zeros (specifying an unstructured file), and *file-type*.<12> is 0, then an even unstructured file is created.

  If *file-type*.<12> is passed as 0, the values of *read-count* and *write-count* are each rounded up to an even number before the operation begins; for example, a *write-count* or *read-count* of 7 is rounded up to 8, and 8 bytes are transferred.

  A file must be positioned to an even byte address; otherwise, FILEINFO or FILE_GETINFO_ returns a file-system error (bad address).

  If you use the FUP CREATE or the TACL CREATE command to create the file, it creates an even unstructured file by default.

- Insertion-ordered alternate keys

  All of the non-unique alternate keys of a file must have the same duplicate key ordering attribute. That is, a file may not have both insertion-ordered alternate keys and standard (duplicate ordering by primary key) non-unique alternate keys. An insertion-ordered alternate key cannot share an alternate key file with other keys of different lengths, or with other keys which are not insertion-ordered.

  The CREATE procedure returns file error 46 if the rules of usage for insertion-ordered alternate keys are violated.

  When an alternate-key record is updated, the timestamp portion of the key is also updated. Alternate-key records are updated only when the corresponding alternate-key field of the primary record is changed.

  The relative position of an alternate-key record within a set of duplicates may change if a nonrecoverable error occurs during a WRITEUPDATE of the primary record.

  There is a performance penalty for using insertion-ordered duplicate alternate keys. Updates and deletes of alternate-key fields force the disk process to sequentially search the set of alternate-key records having the same *altkeyvalue* until a match is found on the *primarykey-value* portion of the key. (The value of the timestamp field in an alternate key record is not stored in the primary record). The performance cost rises as the number of records having duplicate alternate-key values increases.

  If an insertion-ordered alternate-key file is partitioned, the length of each partition key should be no greater than the total of *altkeytaglen* and *altkeylen*. If the length of any partition key is greater than this sum, then the file system may fail to advise the user of the duplicate key condition (indicated by the warning error code 551).

- Queue files
  - Queue files are created by specifying *file-type* .<9> =1 and *file-type* .<13:15> =3.
  - The *key-sequenced-params* array must be specified. The minimum *key-len* must be 8 bytes, and the *key-offset* must be 0.
  - No *alternate-key-params* can be specified.
  - No *partition-key-params* array can be specified.

Safeguard Considerations

For information on files protected by Safeguard, see the *Safeguard Reference Manual*.

OSS Considerations

- This procedure operates only on Guardian objects. If an OSS file is specified, error 1163 is returned.

Example

```
CALL CREATE ( DISK^FNAME , PRI^EXT , FILE^CODE ,
              SEC^EXT , FILE^TYPE , REC^LEN ,
              DATA^BLK^LEN , KEY^PARAMS );
```

Related Programming Manual

For programming information about the CREATE file-system procedure, see the *Enscribe Programmer's Guide*.

CREATEPROCESSNAME Procedure (Superseded by PROCESSNAME_CREATE_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The CREATEPROCESSNAME procedure returns a unique process name suitable for passing to the NEWPROCESS and NEWPROCESSNOWAIT procedures. This type of naming (as opposed to a predefined process name) is used when the name of a process pair does not need to be known to other processes in the system (for example, in an application run as several process pairs). This process name must be passed in the *name* parameter, not the *file-name* parameter, of the NEWPROCESS procedure.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
CALL CREATEPROCESSNAME ( process-name );              ! o
```

Parameters

process-name output

INT:ref:3

is an array where a system-generated process name returns. The *process-name* parameter is of the form:

```
$zaaaa
```

where

“z” is the letter Z, Y, or X.
“a” represents an alphanumeric character except “o” and “i.”

CREATEPROCESSNAME ensures that the character position after the last “a” is a blank.

Condition Code Settings

< (CCL) indicates that the address passed for *process-name* is out of bounds.

= (CCE) indicates the CREATEPROCESSNAME was successful.

> (CCG) indicates there were no unused names in the reserved name space ($X*name*, $Y*name*, and $Z*name*, where *name* is 1 through 4 alphanumeric characters) for CREATEPROCESSNAME to use.

Considerations

- Process names and CREATEPROCESSNAME

  You use names created by CREATEPROCESSNAME when the process must be named, but the name of that process does not need to be predefined, that is, known by any other process or process pair.

Note. Calling CREATEPROCESSNAME does not create a process or enter the process name into the DCT.

- HP reserved process names

  The operating system reserved process name space includes these names: $X*name*, $Y*name*, and $Z*name*, where *name* is 1 through 4 alphanumeric characters. Do not use names of this form in any applications.

- Creating pseudo-temporary disk file names

The CREATEPROCESSNAME procedure is also useful for creating “pseudo-temporary” disk file names. You might use this type of naming when two processes want to use the same file, but each opens the file exclusively.

If a standard temporary file name is used, the file is purged when the first process closes it because there are no other opens for the file. The second process is then unable to access the file. An example of using CREATEPROCESSNAME:

```
INT .TEMP^FNAME[0:11] := ["$VOL1 ", 9 * ["   "]];
      .
      .
CALL CREATEPROCESSNAME ( TEMP^FNAME[4] ); ! returns $zddd
TEMP^FNAME[4].<0:7> := "Z";          ! makezzdaa subvol
TEMP^FNAME[8] ':=' TEMP^FNAME[4] FOR 4;  ! make file name
CALL CREATE ( TEMP^FNAME );
IF < THEN ... ; ! error.
       .
       .
```

The name of the file in the TEMP^FNAME array is:

```
$VOL1   Zzdaa    Zzdaa
```

CREATEREMOTENAME Procedure (Superseded by PROCESSNAME_CREATE_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The CREATEREMOTENAME procedure supplies a process name that is unique for the specified system in a network. (This process name goes into the *name* parameter, not the *file-name* parameter, of the NEWPROCESS[NOWAIT] procedure.)

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
CALL CREATEREMOTENAME ( name                    ! o
                       ,sysnum );               ! i
```

Parameters

name output

INT:ref:3

is an array where CREATEREMOTENAME returns a system-generated process name (in local form) that is unique for the designated system. The *name* parameter is of the form:

```
$zaaaa
```

where

"z" is the letter Z, Y, or X.
"a" represents an alphanumeric character except "o" and "i."

CREATEREMOTENAME ensures that the character position after the last "a" is a blank.

sysnum input

INT:value

is a value that specifies the system number for which the process name is to be created.

Condition Code Settings

< (CCL) indicates that the remote destination control table (DCT) could not be accessed or the address passed for *name* is out of bounds.

= (CCE) indicates that CREATEREMOTENAME was successful.

> (CCG) indicates there were no unused names in the reserved name space ($X*name*, $Y*name*, and $Z*name*, where *name* is 1 through 4 alphanumeric characters) for CREATEREMOTENAME to use.

Considerations

- Remote process name characteristics

  CREATEREMOTENAME creates a process name in local form. This name can be passed directly to the NEWPROCESS[NOWAIT] procedure as the *name* parameter in order to create a remote process having that name. It is unnecessary to append a system name to the process name since the physical location of the program file specified in the NEWPROCESS *file-name* includes the system number.

- HP reserved process names

  The operating system reserved process name space includes these names: $X*name*, $Y*name*, and $Z*name*, where *name* is 1 through 4 alphanumeric characters. Do not use names of this form in any applications.

- Remote system DCT

  The creation of a process name does not create a process or make an entry in the remote system's DCT.

Example

```
CALL CREATEREMOTENAME ( NAME , SYS^NUM );
```

CREATORACCESSID Procedure (Superseded by PROCESS_GETINFOLIST_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The CREATORACCESSID procedure is used to obtain the creator access ID (CAID) of the process that created the calling process.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
creator-access-id := CREATORACCESSID;
```

Parameters

creator-access-id returned value

INT

returns the creator access ID (CAID) of the caller's creator in this form:

<0:7> group number {0:255}

<8:15> member number {0:255}

Considerations

- Process access ID (PAID) compared with creator access ID (CAID)

  For a given process, an access ID is a word in the process control block (PCB) that contains a group number in the left byte and a member number in the right byte. There are two access IDs.

  The creator access ID (CAID) is returned from the CREATORACCESSID procedure and identifies the user who created the process. It is normally used, often with the PAID, for security checks on interprocess operations such as stopping a process or creating a backup for a process.

  The process access ID (PAID) is returned from the PROCESSACCESSID procedure and is used to determine whether the process can make requests to the system, for example, to open a file or to stop a process.

  The PAID and the CAID usually differ only when a process is run from a program file that has the PROGID attribute set. This attribute is usually set with the File Utility Program (FUP) SECURE command and PROGID option. In such a case, the process access ID returned by PROCESSACCESSID is the same as the NonStop operating system used ID of the program file's owner.

  Both the PAID and the CAID are returned from the PROCESS_GETINFO[LIST]_ procedures. See the *Guardian Programmer's Guide* for information about process access IDs.

Example

```
CREATOR^ID := CREATORACCESSID;
```

Related Programming Manual

For more information about the creator accessor ID (CAID), see the *Guardian User's Guide*.

CRTPID_TO_PROCESSHANDLE_ Procedure

Summary

The CRTPID_TO_PROCESSHANDLE_ procedure converts a process ID (CRTPID) to the corresponding process handle. For information about process IDs and process handles, see Appendix D, File Names and Process Identifiers.

Syntax for C Programmers

```
#include <cextdecs(CRTPID_TO_PROCESSHANDLE_ )>

short CRTPID_TO_PROCESSHANDLE_ ( short *process-id
                                ,short *processhandle
                                ,[ short *pair-flag ]
                                ,[ __int32_t node-number ] );
```

- CEXTDECS (through the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
error := CRTPID_TO_PROCESSHANDLE_ ( process-id             ! i
                                   ,processhandle          ! o
                                   ,[ pair-flag ]          ! o
                                   ,[ node-number ] );     ! i
```

Parameters

error returned value

INT

returns a file-system error number indicating the outcome of the operation.

process-id input

INT .EXT:ref:4

specifies the process ID (CRTPID) to be converted. If *process-id* does not include a node number, the caller's node is assumed.

processhandle output

INT .EXT:ref:10

returns the process handle of the process designated by *process-id*.

pair-flag output

INT .EXT:ref:1

returns a value of 1 if

- *cpu* and *pin* value in *process-id* is set to -1
- *cpu* and *pin* value in *process-id* is set to blanks (“ “)

It returns a value of 0 otherwise.

node-number input

INT(32):value

if present and not equal to -1D, and if *process-id* is not in network form, identifies the node on which the process identified by *process-id* resides. If omitted or equal to -1D, the caller’s node is assumed.

Considerations

- When converting the process ID process, CRTPID_TO_PROCESSHANDLE_ looks up the process in a system table and it might send a system message. An error 14 is returned if the process does not exist.
- This procedure does not return information on a named process that is reserved for future use and is not started.

Related Programming Manual

For programming information about the CRTPID_TO_PROCESSHANDLE_ procedure, see the *Guardian Application Conversion Guide*.

CURRENTSPACE Procedure (Superseded)

Summary

Note. This procedure cannot be called by native processes. Although this procedure is supported for TNS processes, it should not be used for new development.

The CURRENTSPACE procedure returns the ENV register (as saved in the stack marker) and a string (in ASCII) containing the space ID of the caller.

Syntax for C Programmers

```
#include <cextdecs(CURRENTSPACE)>

short CURRENTSPACE ( [ char *ascii-space-id ] );
```

Syntax for TAL Programmers

```
stack-env := CURRENTSPACE [ ( ascii-space-id ) ];   ! o
```

Parameters

stack-env returned value

INT

is the calling procedure's space ID in the stack-marker ENV register format.

```
ENV.<4>          ! library bit
ENV.<7>          ! system code bit
ENV.<11:15>      ! space ID bits
```

For more information about space identifiers and the details of these bits, see the *System Description Manual* appropriate for your system.

ascii-space-id output

STRING:ref:5

is an ASCII string in the form:

map.<#>

where

map is one of these:

UC indicates user code

UL indicates user library

SC indicates system code

SL indicates system library

<#> is the octal space number in ASCII.

for example:

```
UC.01  or  SL.33
```

Related Programming Manual

For information about the CURRENTSPACE procedure, see the appropriate *System Description Manual* for your system.

4 Guardian Procedure Calls (D-E)

This section contains detailed reference information for all user-accessible Guardian procedure calls beginning with the letters D through E. Table 4-1 lists all the procedures in this section.

Table 4-1. Procedures Beginning With the Letters D Through E (page 1 of 2)

Table 4-1. Procedures Beginning With the Letters D Through E (page 2 of 2)

DAYOFWEEK Procedure

Summary

The DAYOFWEEK procedure takes a 32-bit Julian Day Number and returns the corresponding day of the week.

Syntax for C Programmers

```
#include <cextdecs(DAYOFWEEK)>

short DAYOFWEEK ( __int32_t julian-day-num );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
day := DAYOFWEEK ( julian-day-num );          ! i
```

Parameters

day returned value

INT

is the code for the day of week, as follows: 0 = Sunday, 1 = Monday, ..., 6 = Saturday. If *day* is -1, then the *julian-day-num* was negative.

julian-day-num input

INT(32):value

contains the Julian Day Number for which the day of the week is desired.

Example

```
INT day
INT(32) JDN := 2435012D;
      .
      .
day := DAYOFWEEK ( JDN );
IF day < 0 THEN ...
```

DEALLOCATESEGMENT Procedure (Superseded by SEGMENT_DEALLOCATE_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The DEALLOCATESEGMENT procedure deallocates an extended data segment when it is no longer needed by the calling process.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
CALL DEALLOCATESEGMENT ( segment-id                ! i
                        ,[ flags ] );              ! i
```

Parameters

segment-id input

INT:value

is the segment number of the segment, as specified in the call to ALLOCATESEGMENT that created it.

flags input

INT:value

if present, has the form:

<0:14>
must be 0.

<15> 1 indicates that dirty pages in memory are not to be copied to the swap file (see ALLOCATESEGMENT Procedure (Superseded by SEGMENT_ALLOCATE_ Procedure)).

0 indicates that dirty pages in memory are to be copied to the swap file.

This parameter is ignored if the swap space was allocated using the Kernel-Managed Swap Facility (KMSF).

The default is 0.

Condition Code Settings

< (CCL) Segment not deallocated—an invalid segment ID was supplied or the specified segment is currently in use by the operating system; for example, an outstanding nowait I/O operation using a buffer in the segment has not been completed by a call to AWAITIOX.

= (CCE) Segment deallocated.

> (CCG) Segment deallocated, but an I/O error occurred writing dirty pages to the segment's permanent swap file.

Considerations

- The *flags* parameter

  The *flags*.<15> = 1 option is used to improve performance when the swap file is either a permanent file or a temporary file that is opened concurrently by another application. Following the DEALLOCATESEGMENT call, the contents of the swap file are unpredictable.

  If the DEALLOCATESEGMENT call causes a purge of a temporary file or the DEALLOCATESEGMENT call deallocates swap space managed by the Kernel-Managed Swap Facility (KMSF), the operating system does not write the dirty pages (that is, pages that are being used) out to the file.

- Breakpoints

  Before deallocating a segment, this procedure removes all memory access breakpoints set in that segment.

- Segment deallocation

  When a segment is deallocated, the swap file end of file (EOF) is set to the larger of (1) the EOF when the file is opened by ALLOCATESEGMENT or (2) the end of the highest numbered page that is written to the swap file. All file extents beyond the EOF that did not exist when the file was opened are deallocated.

- Shared segments

  A shared segment remains in existence until it has been deallocated by all the processes that allocated it.

Example

```
CALL DEALLOCATESEGMENT ( SEGMENT^ID );
IF <> THEN ...

! SEGMENT^ID refers to the segment number specified
! in the call to ALLOCATESEGMENT.
```

DEBUG Procedure

Summary

The DEBUG procedure invokes the debugging facility on the calling process.

The operating system provides a debugging facility that responds to debug events by passing control to one of two debugging utilities: Debug or the Inspect debugger. Debug is a low-level debugger. The Inspect debugger is an interactive symbolic debugger that lets you control program execution, display values, and modify values in terms of source-language symbols.

Syntax for C Programmers

```
#include <cextdecs(DEBUG)>

void DEBUG ();
```

Syntax for TAL Programmers

```
CALL DEBUG;
```

Considerations

- While a process is in the debug state, you can interactively display and modify the contents of the process's registers, the process's data area, and set other breakpoints. To debug a program, you must have EXECUTE access to run the program and read access to the program object file.
- In addition to placing an explicit call to the DEBUG procedure in the source program, you can force a process into the debug state by:

  - Starting the process using the command interpreter's RUND (RUN DEBUG) command. The process enters the debug state before the first instruction of the MAIN procedure executes.
  - Starting the process with a call to PROCESS_CREATE_, PROCESS_SPAWN_, NEWPROCESS, NEWPROCESSNOWAIT, OSS `tdm_fork()`, OSS `tdm_spawn()` or one of the OSS `tdm_exec` set of functions, and setting the appropriate debug option. The process enters the debug state before the first instruction of the MAIN procedure executes.
  - Starting the process from the command interpreter. While the process is executing, press the BREAK key. The command interpreter returns to the command input mode. Find the *`cpu,pin`* of the process, and type in DEBUG *`cpu,pin`*.
  - Specifying a breakpoint when a process is in the debug state. When that breakpoint is hit, the process enters the debug state.
- You can use the Inspect debugger by setting the Inspect attribute associated with a process. The value of a process's Inspect attribute can be set with:
  - The ?INSPECT or ?SAVEABEND TAL compiler directive
  - The nld -SET INSPECT or -SET SAVEABEND commands during a linking session
  - The Binder SET INSPECT or SET SAVEABEND commands during a binding session
  - The TACL SET INSPECT command before the RUN command that starts the process
  - The INSPECT parameter of the RUN command that starts the process
  - The appropriate option in the call to PROCESS_CREATE_, PROCESS_SPAWN_, NEWPROCESS, NEWPROCESSNOWAIT, OSS `tdm_fork()`, OSS `tdm_spawn()` or one of the OSS `tdm_exec` set of functions, that starts the process.
- Processes inherit the Inspect attribute from their ancestor processes.

OSS Considerations

When used on an OSS process, DEBUG forces the process into the Inspect debugger.

To debug an OSS process, one of these must be true:

- The calling process must have appropriate privilege; that is, it must be locally authenticated as the super ID on the system where the target process is executing.
- All these apply:
  - The caller's effective user ID is the same as the saved user ID of the target process.

- The caller has sufficient “nonremoteness”; that is, the caller is locally authenticated, or the target process is remotely authenticated and the caller is authenticated from the viewpoint of the system where the target process is executing.
- The caller has read access to the program file and any library files.
- The program does not contain PRIV or CALLABLE routines.
- The target is not a system process.

Only program file owners and users with appropriate privileges are able to debug programs that set the user ID.

Related Programming Manual

For information about the Debug facility, see the *Debug Manual*. For information about the Inspect debugger, see the *Inspect Manual*.

DEBUGPROCESS Procedure (Superseded by PROCESS_DEBUG_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The DEBUGPROCESS procedure invokes the debugging facility on a process.

The operating system provides a debugging facility that responds to debug events by passing control to one of two debugging utilities: Debug or the Inspect debugger. Debug is a low-level debugger. The Inspect debugger is an interactive symbolic debugger that lets you control program execution, display values, and modify values in terms of source-language symbols.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
CALL DEBUGPROCESS ( process-id                  ! i
                  ,error                        ! o
                  ,[ term ]                     ! i
                  ,[ now ] );                   ! i
```

Parameters

process-id input

INT:ref:4

is a 4-word array containing the process ID of the process to be debugged, where:

[0:2]	Process name or creation timestamp
[3].<0:3>	Reserved
[3].<4:7>	processor number where the process is executing
[3].<8:15>	PIN assigned by the operating system to identify the process in the processor

Note that the process ID can be in a timestamp or a local or remote named format.

error output

INT:ref:1

returns a file-system error number indicating the outcome of the process debug attempt. Possible values include these:

0	No error.
11	The specified process does not exist.
13	Invalid name. This error can occur when the supplied process ID is improperly formed.
14	The supplied process ID references an LDEV that does not exist.
18	The specified system is not known.
22	Parameter or buffer out of bounds.
29	Missing parameter.
48	Security violation. The caller does not have read and execute access to the program file, or the caller specified *now* = 1 without having a process access ID (PAID) equal to the super ID (255,255).
190	*term* (or the caller's home terminal if *term* was not specified) is not device type 6.
201	Unable to communicate over this path.

240-249 Network errors.

250 All paths to the specified system are down.

590 Bad parameter value. This error can occur when the supplied process ID is improperly formed.

term input

INT:ref:12

is the name of the debug home terminal. If omitted, the caller's home terminal is used.

now input

INT:value

The caller's process access ID (PAID) must be the super ID (255, 255) to use this parameter.

If you supply 1, the process should be debugged immediately (even if it is currently executing privileged code). If omitted, the normal debug sequence is executed.

Considerations

DEBUGPROCESS cannot be used on a high-PIN unnamed process. However, it can be used on a high-PIN *named* process or process pair; *process-id*[3] must then contain either -1 or two blanks.

To invoke the debug facility on a high-PIN unnamed process, use the PROCESS_DEBUG_ procedure.

OSS Considerations

When used on an OSS process, DEBUGPROCESS forces the process into the Inspect debugger. You can change the home terminal by specifying a valid value in the *term* parameter procedure. Note that the home terminal is often the same device as the controlling terminal.

To debug an OSS process, one of these must be true:

- The calling process must have appropriate privilege; that is, it must be locally authenticated as the super ID on the system where the target process is executing.
- All these apply:
  - The caller's effective user ID is the same as the saved user ID of the target process.
  - The caller has sufficient "nonremoteness"; that is, the caller is locally authenticated, or the target process is remotely authenticated and the caller is authenticated from the viewpoint of the system where the target process is executing.

- The caller has read access to the program file and any library files.
- The program does not contain PRIV or CALLABLE routines.
- The target is not a system process.
- The *now* parameter is not specified.

Only program file owners and users with appropriate privileges are able to debug programs that set the user ID.

DEFINEADD Procedure

Summary

This procedure adds a DEFINE to the calling process's context using the attributes in the working set. It can be used to replace an existing DEFINE with the attributes in the working set.

Syntax for C Programmers

```
#include <cextdecs(DEFINEADD)>

short DEFINEADD ( constchar *define-name
                 ,[ short replace ]
                 ,[ short _near *checknum ] );
```

Syntax for TAL Programmers

```
error := DEFINEADD ( define-name              ! i
                    ,[ replace ]              ! i
                    ,[ checknum ] );          ! o
```

Parameters

error returned value

INT

indicates the outcome of the call:

0	Add was successful
2049	A syntax error occurred in name
2050	Define already exists
2051	Define does not exist
2052	Unable to obtain file-system buffer space
2053	Unable to obtain physical memory
2054	Bounds error in *define-name*
2057	Working set is incomplete, a required attribute is missing.
2058	Working set is inconsistent. Two or more attributes have conflicting values. The *checknum* parameter identifies the consistency check that failed.
2059	Working set is invalid
2066	Missing parameter
2069	The DEFMODE of the process does not permit the addition of the DEFINE

For other error values associated with DEFINEs, see the *Guardian Procedure Errors and Messages Manual*.

define-name input

STRING .EXT:ref:24

is the 24-byte array that contains the name of the DEFINE to be added or replaced in the working set. The name is left-justified and padded on the right with blanks. Trailing blanks are ignored.

replace input

INT:value

if present and has a value of 1, then the attributes of the DEFINE that is named by *define-name* are replaced with the attributes in the working set.

checknum output

INT:ref:1

contains the number of the consistency check that failed when 2058 is returned in *error*. For a list of DEFINE consistency check numbers, see the *Guardian Procedure Errors and Messages Manual*.

Considerations

- If an error occurs, the DEFINE is not created or replaced.
- If the replace option is used, the named DEFINE must exist.
- The context-change count is incremented each time procedure DEFINEADD is invoked and a consequent change to the process's context occurs. If an error occurs, the count is not incremented.

Example

```
STRING .EXT define^name[0:23];
          .
          .
define^name ':=' ["=mydefine                "];
error := DEFINEADD( define^name, 1 );
IF error <> DEOK THEN ... ;
```

Related Programming Manual

For programming information about the DEFINEADD procedure, see the *Guardian Programmer's Guide*.

DEFINEDELETE Procedure

Summary

This procedure allows the caller to delete a DEFINE from the calling process's context.

Syntax for C Programmers

```
#include <cextdecs(DEFINEDELETE)>

short DEFINEDELETE ( constchar *define-name );
```

Syntax for TAL Programmers

```
error := DEFINEDELETE ( define-name );        ! i
```

Parameters

error returned value

INT

indicates the outcome of the call:

0	Add was successful
2049	A syntax error occurred in name
2051	Define does not exist
2052	Unable to obtain file-system buffer space
2054	Bounds error in *define-name*
2066	Missing parameter

For other error values associated with DEFINEs, see the *Guardian Procedure Errors and Messages Manual*.

define-name input

STRING .EXT:ref:24

is the 24-byte array that contains the name of the DEFINE to be deleted. The name is left-justified and padded on the right with blanks. Trailing blanks are ignored.

Considerations

- If an error occurs, the DEFINE is not deleted.
- The context-changes count is incremented each time DEFINEDELETE is invoked and a consequent change to the process's context occurs. The count is incremented by one even if more than one DEFINE is deleted.

Example

```
STRING .EXT define^name[0:23];
         .
         .
define^name ':=' ["=mytape                    "];
error := DEFINEDELETE ( define^name );
IF error <> DEOK THEN ... ;
```

Related Programming Manual

For programming information about the DEFINEDELETE procedure, see the *Guardian Programmer's Guide*.

DEFINEDELETEALL Procedure

Summary

This procedure allows the caller to delete all DEFINEs from the calling process's context.

Syntax for C Programmers

```
#include <cextdecs(DEFINEDELETEALL)>

short DEFINEDELETEALL ();
```

Syntax for TAL Programmers

```
CALL DEFINEDELETEALL;
```

Considerations

- If an error occurs, the DEFINE is not deleted.
- The context-changes count is incremented each time DEFINEDELETEALL is invoked and a consequent change to the process's context occurs. The count is incremented by one even if more than one DEFINE is deleted.
- The =_DEFAULTS DEFINE cannot be deleted and is bypassed by this procedure.

Related Programming Manual

For programming information about the DEFINEDELETEALL procedure, see the *Guardian Programmer's Guide*.

DEFINEINFO Procedure

Summary

This procedure returns selected information about a DEFINE.

Syntax for C Programmers

```
#include <cextdecs(DEFINEINFO)>

short DEFINEINFO ( constchar *define-name
                  ,char *class
                  ,char *attribute-name
                  ,char *value-buf
                  ,short value-buf-len
                  ,short _near *value-len );
```

Syntax for TAL Programmers

```
error := DEFINEINFO ( define-name                ! i
                     ,class                      ! o
                     ,attribute-name             ! o
                     ,value-buf                  ! o
                     ,value-buf-len              ! i
                     ,value-len );               ! o
```

Parameters

error returned value

INT

indicates the outcome of the call:

0	Success
2049	A syntax error occurred in name

2051 DEFINE not found

2052 Unable to obtain file-system buffer space

2054 Bounds error on parameter

2066 Parameter missing

For other error values associated with DEFINEs, see the *Guardian Procedure Errors and Messages Manual*.

define-name input

STRING .EXT:ref:24

is the 24-byte array that contains the name of the DEFINE to be used by the procedure. The name is left-justified and padded on the right with blanks. Trailing blanks are ignored.

class output

STRING .EXT:ref:16

returns the character string that names the class of the DEFINE (that is, names the value of the CLASS attribute of the DEFINE). The name is left-justified and blank-filled. It is limited to 16 characters.

attribute-name output

STRING .EXT:ref:16

is the name of an attribute; the specific attribute that is returned depends on the CLASS attribute of the DEFINE. The name is left-justified and blank-filled. These attributes names are returned:

This attribute…	Is returned for this CLASS attribute of the DEFINE
FILE	CLASS MAP
LOC	CLASS SPOOL
SCRATCH	CLASS SORT and CLASS SUBSORT
SUBVOL	CLASS CATALOG
SUBVOL0	CLASS SEARCH
VOLUME	CLASS TAPE, CLASS TAPECATALOG and CLASS DEFAULTS

value-buf output

STRING .EXT:ref:*

is the data array provided by the calling program to return the value of an attribute. This attribute depends upon the class of the DEFINE. The value will be in external form, suitable for display. If the value is a file name, it is fully qualified.

value-buf-len input

INT:value

is the length of the array *value-buf* in bytes.

value-len output

INT:ref:1

gives the actual size of the external representation for the value. If greater than *value-buf-len*, then only *value-buf-len* bytes have been transferred and truncation has occurred. An absent attribute is indicated by a length of -1.

Considerations

This procedure is designed to support the short form of the command interpreter INFO command.

Example

```
STRING .EXT define^name[0:23];
STRING .EXT class^name[0:15];
STRING .EXT attr^name[0:15];
STRING .EXT value^buf[0:n];
INT  value^buf^len;
INT  value^len;
         .
         .
define^name ':=' ["=mytape                  "];
value^buf^len := n;
error := DEFINEINFO( define^name, class^name, attr^name,
                     value^buf, value^buf^len, value^len );
IF error <> DEOK THEN ... ;
```

DEFINELIST Procedure

Summary

The DEFINELIST procedure is used only when the application process is acting as a supervisor or tributary station in a centralized multipoint configuration.

- Within a supervisor station, DEFINELIST specifies the station addresses of each tributary station that the application process wishes to communicate with.
- Within a tributary station, DEFINELIST specifies the station addresses that the particular line responds to.

The addresses are in the form of a “station list” array whose name passes to the DEFINELIST procedure by way of the DEFINELIST calling sequence.

Syntax for C Programmers

```
#include <cextdecs(DEFINELIST)>

_cc_status DEFINELIST ( short filenum
                      ,short _near *address-list
                      ,short address-size
                      ,short num-entries
                      ,short polling-count
                      ,short polling-type );
```

- The function value returned by DEFINELIST, which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL DEFINELIST ( filenum                                ! i
                 ,address-list                           ! i
                 ,address-size                           ! i
                 ,num-entries                            ! i
                 ,polling-count                          ! i
                 ,polling-type );                        ! i
```

Parameters

filenum input

INT:value

is the name of the one-word integer variable specified in the call to FILE_OPEN_ or OPEN that opened the line.

address-list input

INT:ref:*

is the name of an integer array containing either:

- Polling addresses and selection addresses (for a description of this array, see the *Envoy Byte-Oriented Protocols Reference Manual*)

- one or more station addresses (for a description of this array, see the *EnvoyACP/XF Reference Manual*)

address-size input

INT:value

specifies the size, in words, of an entry in the *station-list* array. Note that the entry size varies somewhat from one protocol to another.

num-entries input

INT:value

specifies the total number of entries in the *station-list* array.

polling-count input

INT:value

specifies the number of polling addresses in the *station-list* array. This parameter has no meaning when used for EnvoyACP bit-oriented protocols.

polling-type input

INT:value

For a supervisor station, specifies the number of times that the tributary stations with polling addresses in the *station-list* array are to be polled when the line is in the control state, and the supervisor station issues a call to READ:

0 Poll continuously

1-127 Number of polling cycles

For tributary stations, this parameter has no functional effect; a dummy argument must still be supplied, however, for each station except Envoy's multipoint tributary. In this case, the *polling-type* can be:

0 RVI (reverse interrupt)

1 WACK (wait for acknowledgment)

2 NAK (negative acknowledge)

Condition Code Settings

< (CCL) indicates that an error occurred (call FILE_GETINFO_ or FILEINFO).

= (CCE) indicates that the DEFINELIST procedure was executed successfully.

> (CCG) does not return from DEFINELIST.

Considerations

Call DEFINELIST after the call to FILE_OPEN_ or OPEN but before the first call to READ or WRITE.

Related Programming Manual

For programming information about the DEFINELIST procedure, see the data communication manuals.

DEFINEMODE Procedure

Summary

This procedure allows the caller to control the use of DEFINEs (the DEFINE mode of the process). See the *Guardian Programmer's Guide* for details on the DEFINE mode and its effects.

Syntax for C Programmers

```
#include <cextdecs(DEFINEMODE)>

short DEFINEMODE ( [ short new-value ]
                 ,[ short _near *old-value ] );
```

Syntax for TAL Programmers

```
error := DEFINEMODE ( [ new-value ]              ! i
                    ,[ old-value ] );            ! o
```

Parameters

error returned value

INT

indicates the outcome of the call:

0 Success

2067 The value supplied in *new-value* is invalid

new-value input

INT:value

is 0 to disable DEFINEs, and 1 to enable DEFINEs. Note that when setting this value you should see that the desired DEFINEs are propagated and usable. For further information, see the *Guardian Programmer's Guide*.

old-value output

INT:ref:1

if present, returns the previous status of the DEFINE mode: 0 (OFF) or 1 (ON).

Considerations

- The *new-value* and *old-value* parameters correspond to the DEFMODE attribute of a process:

  DEFMODE OFF corresponds to value 0;
  DEFMODE ON corresponds to value 1.

- If *new-value* is not supplied, the call to DEFINEMODE does not change the current value of DEFINE mode.
- When a process is created, the DEFINE mode for the new process can be supplied as an option to PROCESS_CREATE_, PROCESS_SPAWN_, NEWPROCESS, NEWPROCESSNOWAIT, OSS `tdm_fork()`, OSS `tdm_spawn()` or one of the OSS `tdm_exec` set of functions, or to the command interpreter RUN command. The default is the DEFINE mode of the caller of the procedure that creates the new process or that of the command interpreter.
- The DEFMODE of a primary process is checkpointed to the corresponding backup process whenever the primary process calls CHECKPOINT or CHECKPOINTMANY to checkpoint the data stack.
- For details on DEFINE mode and its effects, see the *Guardian Programmer's Guide*.

Example

```
INT previous^use;
LITERAL define^mode = 1;
          .
          .
          .
error := DEFINEMODE( define^mode, previous^use );
IF error <> DEOK THEN ...

! The above statements enable DEFINE use and return
```

```
! the previous DEFINE mode in the variable
! previous^use.
```

Related Programming Manual

For programming information about the DEFINEMODE procedure, see the *Guardian Programmer's Guide*.

DEFINENEXTNAME Procedure

Summary

This procedure returns the name of the DEFINE that follows the specified DEFINE (in ASCII order).

Syntax for C Programmers

```
#include <cextdecs(DEFINENEXTNAME)>

short DEFINENEXTNAME ( char *define-name );
```

Syntax for TAL Programmers

```
error := DEFINENEXTNAME ( define-name );      ! i,o
```

Parameters

error returned value

INT

indicates the outcome of the call:

0	Successful
2049	A syntax error occurred in name
2051	DEFINE not found
2052	Unable to obtain file-system buffer space
2054	Parameter address is bad
2066	Missing parameter
2060	No more DEFINEs

For other error values associated with DEFINEs, see the *Guardian Procedure Errors and Messages Manual*.

define-name input, output

STRING .EXT:ref:24

on input, is a DEFINE name. It need not name an existing DEFINE. The name must be left-justified and padded on the right with blanks. Trailing blanks are ignored.

on output, is the name of the DEFINE following the input DEFINE name in the process context (in ASCII order); if *define-name* is blank on input, the name of the first DEFINE is returned.

Considerations

- To obtain the name of the very first DEFINE in the process context, *define-name* must be blanks.
- On output, *define-name* is either a valid existing DEFINE (on success) or is unchanged (on failure).

Example

In this example, DEFINENEXTNAME returns “=my^output” which directly follows “=my^input” in ASCII order. These DEFINEs were created previously.

```
STRING .file^name [0:36];

file^name ':=' ["=my^input"];
error := DEFINENEXTNAME( file^name );
IF error <> DEOK THEN ...
```

DEFINEPOOL Procedure (Superseded by POOL_* Procedures)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development. *POOL procedures are replaced by POOL_* procedures. There is no one-for-one replacement.

The DEFINEPOOL procedure designates a portion of a user's stack or an extended data segment for use as a pool.

Syntax for C Programmers

```
#include <cextdecs(DEFINEPOOL)>

short DEFINEPOOL ( short *pool-head
                  ,short *pool
                  ,__int32_t pool-size );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
status := DEFINEPOOL ( pool-head              ! o
                      ,pool                   ! i
                      ,pool-size );           ! i
```

Parameters

status returned value

INT

returns a status word having one of these values:

0	No error
1	Bounds error on *pool-head*
2	Bounds error on *pool*
3	Invalid *pool-size*
4	*pool-head* and *pool* overlap.
5	*pool-head* is not word-aligned.
6	*pool* is not word-aligned.

pool-head output

INT .EXT:ref:19

is a 19-word array to be used as the pool header; GETPOOL and PUTPOOL use this array to manage the pool. An even-byte address must be specified.

pool input

INT .EXT:ref:*

specifies the address of the first word of the memory space to be used as the pool. An even-byte address must be specified. The address of the actual beginning of the pool might be adjusted for alignment.

pool-size input

INT(32):value

specifies the size of the pool in bytes. This number must be a multiple of 4 bytes and cannot be less than 32 bytes or greater than 127.5 megabytes (133,693,440 bytes). The address of the end of the pool is always equal to the address specified for the *pool* parameter plus *pool-size*. Pool space overhead and adjustments for alignment do not cause the pool to extend past this boundary.

△ **Caution.** If a privileged process calls DEFINEPOOL and supplies an odd-byte address for the *pool* or the *pool-head* parameter, a processor halt results.

Considerations

- Stack addresses converted to extended addresses

  If *pool-head* or *pool* is in the user data stack, the TAL compiler automatically converts data stack addresses to extended addresses.

- Read-only segments

  If you specify *pool-head* or *pool* in an extended data segment that is allocated as a read-only segment, DEFINEPOOL returns error 1 or 2 (bounds error on *pool-head* or *pool*, respectively).

- Dynamic memory allocation

  Several Guardian procedures support the creation of memory pools and dynamic allocation of variable-sized blocks from the pool. The calling program provides the memory area to be used as the pool and then calls the DEFINEPOOL procedure to initialize a 19-word array, the *pool-header*, that is used to manage the pool. The pool and the pool header can reside in the user data stack or in extended memory. The pool routines accept and return extended addresses that apply to both the stack and extended memory.

  Once the pool is defined, the process can reserve blocks of various sizes from the pool by calling the GETPOOL procedure and can release blocks by calling the PUTPOOL procedure. The program must release one entire block using PUTPOOL; it may not return part of a block or multiple blocks in one PUTPOOL call.

  Be careful to use only the currently reserved blocks of the pool, or the pool structure is corrupted and unpredictable results occur. If multiple pools are

defined, do not return reserved blocks to the wrong pool. For debugging purposes, a special call to GETPOOL checks for pool consistency.

- Pool management methods

  This information is supplied for use in evaluating the appropriateness of using the Guardian pool routines in user application programs and determining the proper size of a pool. Application programs should not depend on the pool data structures, since they are subject to change. The program should use only the procedural interfaces described on these pages.

  The requested block size is rounded up to a multiple of 4 bytes, at a minimum of 28 bytes. This reduces pool fragmentation, but when the program is allocating small blocks, it can waste memory space.

  One extra word is allocated for a boundary tag at the beginning and end of each block; thus, the minimum pool block size is 32 bytes. This tag serves three purposes:

  1. It contains the size of each block so that the program does not need to specify the length of the block when releasing it.
  2. It serves as a check to ensure that the program does not erroneously use more memory than the block contains (although it does not stop the program from overwriting).
  3. It provides for efficient coalescing of adjacent free blocks.

  In GETPOOL, the free block list is searched for the first block sufficiently large enough to satisfy the request. If the free block is at least 32 bytes longer than the required size, it is split into a reserved block and a new free block. Otherwise, the entire free block is used for the request.

  In summary, the pool space overhead on each block can be substantial if very small blocks are allocated. An approximate formula is:

```
ALLOCATED := ($MAX (REQUEST + 7, 32) /4) * 4;
```

  where REQUEST is the original request size in bytes; the allocated blocks are also measured in bytes.

  Although they can also be used to manage the allocation of a collection of equal-sized blocks, these procedures are not recommended for that purpose, because they can consume more processor time and pool memory than user-written routines designed for that specific task.

Example

```
STATUS := DEFINEPOOL ( POOL^HEAD , POOL , 2048D );
```

Related Programming Manual

For programming information about the DEFINEPOOL memory-management procedure, see the *Guardian Programmer's Guide*.

DEFINEREADATTR Procedure

Summary

This procedure allows the caller to obtain the current value of an attribute in a DEFINE in the calling process's context or in the working set. The value of a specific attribute can be read or all the attributes can be sequentially read. The value is returned in an ASCII string form suitable for display.

Syntax for C Programmers

```
#include <cextdecs(DEFINEREADATTR)>

short DEFINEREADATTR( [ constchar *define-name ]
                     ,char *attribute-name
                     ,[ short _near *cursor ]
                     ,char *value-buf
                     ,short value-buf-len
                     ,short _near *value-len
                     ,[ short read-mode ]
                     ,[ short _near *info-word ] );
```

Syntax for TAL Programmers

```
error := DEFINEREADATTR ( [ define-name ]       ! i
                         ,attribute-name        ! i,o
                         ,[ cursor ]            ! i,o
                         ,value-buf             ! o
                         ,value-buf-len         ! i
                         ,value-len             ! o
                         ,[ read-mode ]         ! i
                         ,[ info-word ] );      ! o
```

Parameters

error returned value

INT

indicates the outcome of the call:

0	Successful
2049	A syntax error occurred in name
2051	DEFINE not found
2052	An error occurred when placing PFS in use
2054	Bounds error on parameter
2055	Attribute not supported
2061	No more attributes (see “Considerations”)
2066	Missing parameter

For other error values associated with DEFINEs, see the *Guardian Procedure Errors and Messages Manual*.

define-name input

STRING .EXT:ref:24

is the 24-byte array that contains the name of the DEFINE for the procedure to use. The name is left-justified and padded on the right with blanks. Trailing blanks are ignored.

Omit this parameter to refer to the working set.

attribute-name input, output

STRING .EXT:ref:16

If *cursor* is absent, then this parameter names the attribute whose value is to be returned.

If *cursor* is present, then this is an output parameter. The name is left-justified and blank-filled on output.

cursor input, output

INT:ref:1

is a pointer to the attribute on input. On output, *cursor* points to the sequentially next attribute that is to be read. To read the first attribute, set the *cursor* to 0.

To sequentially read all attributes, do not modify this parameter.

If both *attribute-name* and *cursor* are present, then *cursor* is used to identify the attribute.

value-buf output

STRING .EXT:ref:*

is the data array provided by the calling program to return the value of the attribute. The value is in external form, suitable for display. If the value is a file name, it is fully qualified.

value-buf-len input

INT:value

is the length of the array *value-buf* in bytes.

value-len output

INT:ref:1

gives the actual size of the external representation for the value. If greater than *value-buf-len*, then only *value-buf-len* bytes have been transferred and truncation has occurred. An absent attribute is indicated by a length of -1.

read-mode input

INT:value

is used with *cursor*. It indicates the search mode for the next parameter whose *cursor* value is to be returned.

0 Search present attributes only.

1 Search present plus required attributes that are not present.

2 Search present plus required and optional attributes that are not present.

If *read-mode* is not supplied, 0 is used.

info-word output

INT:ref:1

info-word.<14:15> indicates the type of the attribute:

0 optional

1 defaulted

2 required

info-word.<13> is set if this attribute was involved in an inconsistency at the last check. (For a list of DEFINE consistency check numbers, see DEFINEADD parameter *checknum*.)

Considerations

- This procedure can be used to obtain the value of any attribute in the DEFINE working set, including the CLASS attribute.

- If an error occurs, the contents of the data array are undefined.
- Both *attribute-name* and *cursor* can be present. If *cursor* is present, it is used to "name" the attribute whose value is to be returned, and *attribute-name* returns the name of the attribute.
- When the *cursor* parameter is used, parameter *info-word* is returned even though the attribute can be absent from the DEFINE working set.
- To use the cursor mode, initialize *cursor* to 0 and repeatedly call this procedure without changing *cursor* to sequentially read attributes. The caller should not, for example, set *cursor* to 7 and then call this procedure.
- To implement a command similar to the TACL SHOW DEFINE command, a process would typically call DEFINEREADATTR with *define-name* omitted and with *read-mode* equal to 1; to implement the SHOW DEFINE * command, it would call DEFINEREADATTR with *define-name* omitted and with *read-mode* equal to 2.
- To implement the detailed version of the INFO DEFINE command, command interpreters would call DEFINEREADATTR passing it the *define-name* and with *read-mode* of 0.
- When the cursor option is being used, and the last attribute is read, then *cursor* returns the next attribute number consistent with the *read-mode* parameter. When this attribute is read, 2061 (no more attributes) is returned instead of 0. The 2061 code should be interpreted as success; however, if a process (such as a command interpreter) is calling DEFINEREADATTR in a loop using the cursor option, then code 2061 should be used to terminate the loop.
- *attribute-name* should not be declared as a P-relative array. In general, a reference parameter should not be declared as a P-relative array.

Example

```
LITERAL define^vol^len = 25;           ! value buffer length
STRING .EXT define^name [0:23];
STRING .EXT volume [0:15];             ! attribute name
STRING .EXT volid [0:define^vol^len];  ! value buffer
INT len^read := 0;                     ! len of external rep.
         .
         .
define^name ':=' ["=mytape                  "];
volume ':=' ["volume          "];
volid ':=' "  " & volid[ 0 ] for define^vol^len;
error := DEFINEREADATTR ( define^name, volume,  , volid,
                          define^vol^len, len^read );
IF error <> THEN ...
```

DEFINERESTORE Procedure

Summary

DEFINERESTORE uses a saved version of a DEFINE in the user's buffer to create an active DEFINE. If an active DEFINE of the same name already exists, it can optionally be replaced. The saved DEFINE can also be placed in the working set without its name.

Syntax for C Programmers

```
#include <cextdecs(DEFINERESTORE)>

short DEFINERESTORE ( short *buffer
                     ,[ short options ]
                     ,[ char *define-name ]
                     ,[ short _near *checknum ] );
```

Syntax for TAL Programmers

```
error := DEFINERESTORE ( buffer              ! i
                        ,[ options ]         ! i
                        ,[ define-name ]     ! o
                        ,[ checknum ] );     ! o
```

Parameters

error returned value

INT

is a number that indicates the outcome of the call.

0	Successful
2050	DEFINE already exists and *options*.<15> is 0 or *options* is omitted
2051	DEFINE does not exist and *options*.<15> is 1
2052	Unable to obtain file system buffer space
2053	Unable to obtain physical memory

2054 Bounds error on *buffer*, *define-name* or *checknum* parameter

2055 Invalid attribute in saved DEFINE

2057 DEFINE or working set is incomplete. A required attribute is missing.

2058 DEFINE or working set is inconsistent. Two or more attributes have conflicting values. The *checknum* parameter identifies the consistency check that failed.

2059 DEFINE or working set is invalid

2066 Parameter missing

2067 Attribute contained an invalid value

2068 Saved DEFINE was of invalid CLASS

2069 Attempt to add a DEFINE that does not fall under the current DEFMODE setting

2075 *option*.<0:13> is not 0

2077 *buffer* or *define-name* is in invalid segment

2078 *buffer* does not contain a valid saved DEFINE

For other error values associated with DEFINEs, see the *Guardian Procedure Errors and Messages Manual*.

buffer input

INT .EXT:ref:*

contains the saved form of the DEFINE.

options input

INT:value

indicates whether the saved DEFINE should be restored to the working attribute set or to the active set of DEFINEs. If the latter, it also indicates whether or not the DEFINE replaces an existing DEFINE or is simply added to the active set.

<0:13> are reserved and must be 0

<14> 1 place the saved DEFINE in the working set. If *options*.<14> is 1, then *options*.<15> is ignored.

0 make the saved DEFINE an active DEFINE

<15> 1 replace an existing DEFINE. If a DEFINE of the same name does not exist, an error is returned.

0 add the DEFINE. If a DEFINE of the same name exists, return an error.

If *options* is omitted, the default value of 0 is used; in other words, the saved DEFINE is added to the set of active DEFINEs.

define-name output

STRING .EXT:ref:24

if present, contains the name of the saved DEFINE added to the current context. The name is either the name of the DEFINE when it was saved or the name given the working set when it was saved.

checknum output

INT:ref:1

if present, and the DEFINE is inconsistent, contains the number of the consistency check that failed. For a list of DEFINE consistency check numbers, see the *Guardian Procedure Errors and Messages Manual*.

Considerations

- The buffer must contain a valid internal form of a DEFINE, as created by DEFINESAVE. If the buffer does not appear to contain a valid saved DEFINE, an error is returned and the DEFINE is not added to the current set or to the working set.
- If DEFINERESTORE encounters any error condition while attempting to restore the saved DEFINE to the active set, it does not perform the restore.
- DEFINEs saved by later RVUs of the operating system will be restorable only if the class of the DEFINE is supported on the earlier RVU and the attributes and their values are supported on the earlier RVU.
- If DEFINERESTORE encounters error 2057, 2058, or 2059 (DEFINE invalid, incomplete, or inconsistent) while attempting to restore the saved DEFINE to the working attribute set, it still performs the restore. If it encounters any other errors, however, it leaves the working attribute set unchanged.
- An attempt to restore a saved DEFINE into the active set does not affect the working attribute set or the background set under any circumstances.
- Since the DEFAULTS DEFINE always exists, it cannot be added. If the DEFAULTS DEFINE is saved, the replace option must be used to restore it.

Related Programming Manual

For programming information about the DEFINERESTORE procedure, see the *Guardian Programmer's Guide.*

DEFINERESTOREWORK[2] Procedures

Summary

DEFINERESTOREWORK restores the working set from the background set. The working set is the current set of attributes and their values. A background set is a scratchpad work area used when creating DEFINEs. DEFINERESTOREWORK2 allows a second background working set, saved by DEFINESAVEWORK2, to be restored.

Restoring a background set to a working set does not change the content of the background set.

Syntax for C Programmers

```
#include <cextdecs(DEFINERESTOREWORK)>

short DEFINERESTOREWORK ();


#include <cextdecs(DEFINERESTOREWORK2)>

short DEFINERESTOREWORK2 ();
```

Syntax for TAL Programmers

```
error := DEFINERESTOREWORK[2];
```

Parameters

error returned value

INT

indicates the outcome of the call:

0	Success
2052	Unable to obtain file-system buffer space
2053	Unable to obtain physical memory

For other error values associated with DEFINEs, see the *Guardian Procedure Errors and Messages Manual*.

Related Programming Manual

For programming information about the DEFINERESTOREWORK[2] procedures, see the *Guardian Programmer's Guide*.

DEFINESAVE Procedure

Summary

DEFINESAVE copies an active DEFINE or the current working attribute set into a user buffer. The saved DEFINE can later be made an active DEFINE or be placed into the working set by using DEFINERESTORE.

Syntax for C Programmers

```
#include <cextdecs(DEFINESAVE)>

short DEFINESAVE ( constchar *define-name
                  ,short *buffer ]
                  ,short buflen
                  ,short *deflen
                  ,[ short option ] );
```

Syntax for TAL Programmers

```
error := DEFINESAVE ( define-name       ! i
                     ,buffer            ! o
                     ,buflen            ! i
                     ,deflen            ! o
                     ,[ option ] );     ! i
```

Parameters

error returned value

INT

indicates the outcome of the call:

0	Successful
2049	Syntax error in name

2051 DEFINE not found

2052 Unable to obtain file-system buffer space

2053 Not enough physical memory

2054 Bounds error on *buffer*, *deflen* or *define-name* parameters

2057 DEFINE or working set is incomplete. A required attribute is missing.

2058 DEFINE or working set is inconsistent. Two or more attributes have conflicting values. The *checknum* parameter of the DEFINEADD procedure identifies the consistency check that failed.

2059 DEFINE or working set is invalid

2066 Parameter missing

2075 *option*.<0:14> is not 0

2076 User's buffer is too small

2077 *buffer* or *define-name* is in invalid segment

2079 An attempt to save the working set, but *define-name* is =_DEFAULTS and working set is not class DEFAULTS

For other error values associated with DEFINEs, see the *Guardian Procedure Errors and Messages Manual*.

define-name input

STRING .EXT:ref:24

is a DEFINE name. The name is left-justified and padded on the right with blanks. Trailing blanks are ignored. Depending on the value of *option*, *define-name* contains the name of an existing DEFINE or the name to be given to the working set.

buffer output

INT .EXT:ref:*

is the data array provided by the calling program to contain the saved DEFINE.

buflen input

INT:value

is the length of the array *buffer* in bytes.

deflen output

INT .EXT:ref:1

is the length of the saved DEFINE in bytes.

option input

INT:value

indicates whether the working set or an active DEFINE is to be saved:

<0:14> are reserved and must be 0

<15> 1 save the current working set and name it *define-name*

0 save the active DEFINE named by *define-name*

If *option* is omitted, then the active DEFINE named by *define-name* is saved.

Considerations

- The DEFINE saved in *buffer* is in internal form. You should not modify it. If you change it in any way, DEFINE RESTORE might not be able to restore it.
- If you are saving the working set, *define-name* may contain the name of an active DEFINE; however, the active DEFINE is not saved. Instead, the working set is saved and is given *define-name* as its name in the internal form.
- The working set can be saved if it is inconsistent, invalid or incomplete. A warning is returned in *error*.
- If the user's buffer is too small, *error* will contain 2076 and *deflen* will contain the buffer size required, in bytes. To reduce the possibility of getting error 2076, allocate a buffer of 4096 bytes.

Related Programming Manual

For programming information about the DEFINESAVE procedure, see the *Guardian Programmer's Guide*.

DEFINESAVEWORK[2] Procedure

Summary

DEFINESAVEWORK saves the DEFINE working set in the background set.

DEFINESAVEWORK2 allows a second background working set to be saved.

Syntax for C Programmers

```
#include <cextdecs(DEFINESAVEWORK)>

short DEFINESAVEWORK ();


#include <cextdecs(DEFINESAVEWORK2)>

short DEFINESAVEWORK2 ();
```

Syntax for TAL Programmers

```
error := DEFINESAVEWORK[2];
```

Parameters

error returned value

INT

indicates the outcome of the call:

0	Success
2052	Unable to obtain file-system buffer space
2053	Unable to obtain physical memory

For other error values associated with DEFINEs, see the *Guardian Procedure Errors and Messages Manual*.

Related Programming Manual

For programming information about the DEFINESAVEWORK[2] procedures, see the *Guardian Programmer's Guide*.

DEFINESETATTR Procedure

Summary

This procedure allows the caller to modify the value of an attribute in the working set. The value is supplied in ASCII string form. It is validated, converted into the internal representation and established as the value for the attribute. If the value is a file name or a subvolume name, the default volume information is used to convert the value into the internal form.

This procedure can also be used to reset the value of an attribute to its default value, if one exists, or to delete the attribute from the working set. The attributes of the different DEFINE classes are described in Appendix E, DEFINEs.

Syntax for C Programmers

```
#include <cextdecs(DEFINESETATTR)>

short DEFINESETATTR ( constchar *attribute-name
                     ,[ const char *value ]
                     ,[ short value-len ]
                     ,[ short _near *default-names ] );
```

Syntax for TAL Programmers

```
error := DEFINESETATTR ( attribute-name           ! i
                        ,[ value ]                ! i
                        ,[ value-len ]            ! i
                        ,[ default-names ] );     ! i
```

Parameters

error returned value

INT

indicates the outcome of the call:

0	Successful
2049	A syntax error occurred in name
2052	Unable to obtain file-system buffer space
2055	Attribute not supported
2062	Attribute name too long
2063	A syntax error occurred in default names
2064	The required attribute cannot be reset
2066	Missing parameter
2067	Invalid value

For other error values associated with DEFINEs, see the *Guardian Procedure Errors and Messages Manual*.

attribute-name input

STRING .EXT:ref:16

uniquely identifies an attribute. The name should be left justified and blank-filled.

value input

STRING .EXT:ref:*

is the address of the array that contains the attribute value as an ASCII string (see Appendix E, DEFINEs).

If this parameter is absent, the reset operation is assumed (the attribute is given a default value, if one exists; else the attribute is deleted).

If this parameter is present, then the next parameter must be present.

value-len input

INT:value

is the length of the array *value* in bytes. If -1, the value of the attribute is reset; the attribute is given a default value, if it has one, or the attribute is deleted.

default-names input

INT:ref:8

contains the default volume and subvolume names to be used to convert from the external representation of the value to an internal representation.

[0:3] default volume name. First two bytes can be "*sysnum,*" in which case "$" is omitted from volume name. (blank-filled on right)

[4:7] default subvolume name (blank-filled on right)

Considerations

- To reset an attribute, either the *value* parameter can be omitted, or *value-len* can be -1.
- "Required" attributes cannot be reset. (See the *TACL Reference Manual*.)
- If an error occurs, the contents of the working set are not modified.
- The form of *value*, with respect to quotes, depends on the attribute. The use of quotes should be avoided.

  Quotes can be used with the FILEID, MOUNTMSG, and OWNER attributes. Quotes are discarded from the beginning and end of the string.

Text not enclosed in quotes requires only one quote for a quote mark; text delimited by quotes needs two quotes for a quote mark. The leading and trailing quotes do not count toward the length of the attribute.

- A list of values must have the values separated by commas and must be enclosed in parentheses.
- When CLASS attribute is set (even if the value is not changed), the working set is reinitialized with the attributes of the new class and their default values.
- *attribute-name* should not be declared as a P-relative array. In general, a reference parameter should not be declared as a P-relative array.
- *default-names* should be supplied in certain cases. For more information, see Setting Attributes Using the DEFINESETATTR Procedure in the *Guardian Programmer's Guide*.

Example

```
STRING .EXT labelprocessing [0:15];    ! attribute name
STRING .EXT value [0:15];              ! attribute value
INT .default^names [0:7];
LITERAL value^len = 6;          ! attribute value length
         .
         .
default^names ':=' ["$VOL    MYSUBVOL"];
labelprocessing ':=' ["labels          "];
value ':=' ["bypass"];
error := DEFINESETATTR( labelprocessing, value,
                            value^len, default^names );
IF error <> DEOK THEN ...
```

Related Programming Manual

For programming information about the DEFINESETATTR procedure, see the *Guardian Programmer's Guide*.

DEFINESETLIKE Procedure

Summary

This procedure can be used to initialize the working set with the attributes in an existing DEFINE.

Syntax for C Programmers

```
#include <cextdecs(DEFINESETLIKE)>

short DEFINESETLIKE ( constchar *define-name );
```

Syntax for TAL Programmers

```
error := DEFINESETLIKE ( define-name );          ! i
```

Parameters

error returned value

INT

indicates the outcome of the call:

0	Successful
2049	A syntax error occurred in name
2051	DEFINE not found
2052	Unable to obtain file-system buffer space
2053	Unable to obtain physical memory
2054	Bounds error occurred on *define-name*
2066	DEFINE name is missing

For other error values associated with DEFINEs, see the *Guardian Procedure Errors and Messages Manual*.

define-name input

STRING .EXT:ref:24

is the 24-byte array that contains the name of the DEFINE for the procedure to use. The name is left-justified and padded on the right with blanks. Trailing blanks are ignored.

Considerations

The existing attributes in the working set are deleted. They can be saved in the background set by calling DEFINESAVEWORK before calling this procedure.

Related Programming Manual

For programming information about the DEFINESETLIKE procedure, see the *Guardian Programmer's Guide.*

DEFINEVALIDATEWORK Procedure

Summary

This procedure can be used to check the working set for consistency.

Syntax for C Programmers

```
#include <cextdecs(DEFINEVALIDATEWORK)>

short DEFINEVALIDATEWORK ( short _near *checknum );
```

Syntax for TAL Programmers

```
error := DEFINEVALIDATEWORK ( checknum );          ! o
```

Parameters

error returned value

INT

indicates the outcome of the call:

0	Successful
2057	Working set is incomplete. A required attribute is missing.
2058	Working set is inconsistent. Two or more attributes have conflicting values. The *checknum* parameter identifies the consistency check that failed.
2059	Working set is invalid

For other error values associated with DEFINEs, see the *Guardian Procedure Errors and Messages Manual*.

checknum output

INT:ref:1

contains the consistency check number that failed when 2058 is returned in *error*. See the DEFINEADD parameter *checknum* for a list of DEFINE consistency check numbers.

Considerations

- Subsequent calls to DEFINEREADATTR will return *info-word*.<13> = 1 if the attribute was involved in an inconsistency.
- If the last call to DEFINEREADATTR showed that *info-word*.<13> was set and the DEFINE is currently valid, then a call to DEFINEVALIDATEWORK will clear the flag.
- DEFINEADD invokes this procedure before creating a DEFINE or replacing an existing DEFINE with the working set.
- The command interpreter SHOW command invokes this procedure before calling DEFINEREADATTR.
- DEFINEREADATTR can be used to obtain more information about the attributes in the working set that can be useful to determine why the working set is inconsistent, incomplete or both (invalid).

Related Programming Manual

For programming information about the DEFINEVALIDATEWORK procedure, see the *Guardian Programmer's Guide*.

DELAY Procedure (Superseded by PROCESS_DELAY_ Procedure (H-Series RVUs Only))

Summary

The DELAY procedure permits a process to suspend itself for a timed interval.

Syntax for C Programmers

```
#include <cextdecs(DELAY)>

void DELAY ( __int32_t time-period );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
CALL DELAY ( time-period );                    ! i
```

Parameters

time-period input

INT(32):value

specifies the time period, in 0.01-second units, for which the caller of DELAY is to be suspended.

Considerations

- *time-period* value <= 0D

  A value of less than or equal to 0D results in no delay as such but returns this process's process control block (PCB) to the ready list to give other processes of the same priority a chance to execute.

- Measuring time by the processor clock

  The DELAY procedure measures time according to the internal clock of the processor in which the calling process is executing. Typically, processor time (that is, time as measured by a particular processor) is slightly different from system time; it also varies slightly from processor to processor, because all the processor clocks typically run at slightly different speeds. System time is determined by taking the average of all the processor times in the system.

  When measuring short intervals of time, the difference between processor time and system time is negligible. However, when measuring long intervals of time (such as several hours or more), the difference can be noticeable. For a discussion about measuring long time intervals, see "Considerations" for the SIGNALTIMEOUT procedure.

Example

```
CALL DELAY ( 1000D );          ! suspend for 10 seconds.
```

DELETEEDIT Procedure

Summary

The DELETEEDIT procedure deletes from an EDIT file all lines that have line numbers in a specified range. Upon completion, the current record number is set to the highest line number in the file that is lower than the deleted range, or to -1 if there is no such line.

DELETEEDIT is an IOEdit procedure and can be used only with files that have been opened by OPENEDIT or OPENEDIT_.

Syntax for C Programmers

```
#include <cextdecs(DELETEEDIT)>

short DELETEEDIT ( short filenum
                  ,__int32_t first
                  ,__int32_t last );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
error := DELETEEDIT ( filenum              ! i
                     ,first                ! i
                     ,last );              ! i
```

Parameters

error returned value

INT

returns a file-system error number indicating the outcome of the operation.

filenum input

INT:value

is the number that identifies the open file from which lines are to be deleted.

first input

INT(32):value

specifies 1000 times the line number of the first line in the range of lines to be deleted. If a negative value is specified, the line number of the first line in the file is used.

last input

INT(32):value

specifies 1000 times the line number of the last line in the range of lines to be deleted. If a negative value is specified, the line number of the last line in the file is used.

Example

In this example, DELETEEDIT deletes lines 50 through 100 from the specified file.

```
INT(32) first := 50000D;
INT(32) last := 100000D;
      .
      .
err := DELETEEDIT ( filenumber, first, last );
```

Related Programming Manual

For programming information about the DELETEEDIT procedure, see the *Guardian Programmer's Guide*.

DEVICE_GETINFOBYLDEV_ Procedure (Superseded on G-series RVUs)

Summary

Note. On G-series RVUs, this procedure is supported for compatibility with previous software and should not be used for new development. This procedure cannot obtain all of the physical attributes of a device. For new development, use the CONFIG_GETINFO_BYLDEV_ procedure.

The DEVICE_GETINFOBYLDEV_ procedure obtains the physical and logical attributes of a device. The device is either specified by logical device number or determined by a search.

Syntax for C Programmers

```
#include <cextdecs(DEVICE_GETINFOBYLDEV_)>

short DEVICE_GETINFOBYLDEV_ ( __int32_t ldevnum
                             ,[ short *logical-info ]
                             ,[ short logical-info-maxlen ]
                             ,[ short *logical-info-len ]
                             ,[ short *primary-info ]
                             ,[ short primary-info-maxlen ]
                             ,[ short *primary-info-len ]
                             ,[ short *backup-info ]
                             ,[ short backup-info-maxlen ]
                             ,[ short *backup-info-len ]
                             ,[ __int32_t timeout ]
                             ,[ short options ]
                             ,[ short match-type ]
                             ,[ short match-subtype ]
                             ,[ char *devname ]
                             ,[ short maxlen ]
                             ,[ short *devname-len ]
                             ,[ short *error-detail ] );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.
- The parameter *maxlen* specifies the maximum length in bytes of the character string pointed to by *devname*. The actual length of the name returned in *devname* is returned in *devname-len*. All three of these parameters must either be supplied or be absent.

Syntax for TAL Programmers

```
error := DEVICE_GETINFOBYLDEV_ ( ldevnum                  ! i
                               ,[ logical-info ]          ! o
                               ,[ logical-info-maxlen ]   ! i
                               ,[ logical-info-len ]      ! o
                               ,[ primary-info ]          ! o
                               ,[ primary-info-maxlen ]   ! i
                               ,[ primary-info-len ]      ! o
                               ,[ backup-info ]           ! o
                               ,[ backup-info-maxlen ]    ! i
                               ,[ backup-info-len ]       ! o
                               ,[ timeout ]               ! i
                               ,[ options ]               ! i
                               ,[ match-type ]            ! i
                               ,[ match-subtype ]         ! i
                               ,[ devname:maxlen ]        !
o:i
                               ,[ devname-len ]           ! o
                               ,[ error-detail ] );       ! o
```

Parameters

error returned value

INT

indicates the outcome of the operation. It returns one of these values:

0 Information was successfully returned.

1 (reserved)

2 Parameter error; *error-detail* contains the number of the first parameter found to be in error, where 1 designates the first parameter on the left.

3 Bounds error; *error-detail* contains the number of the first parameter found to be in error, where 1 designates the first parameter on the left.

4 Device not found; *error-detail* contains a file-system error number.

5 Buffer too small. This error applies only to the *devname*:*maxlen* parameter.

ldevnum input

INT(32):value

specifies a logical device number that is used in one of these ways:

- If *options*.<15> is equal to 0, *ldevnum* designates the device for which information is requested.
- If *options*.<15> is equal to 1, the procedure begins a search of devices starting with the logical device number immediately following *ldevnum*.

See the *options* parameter on page 4-53.

On G-series RVUs, the logical device number of a device can change whenever a device is configured or the system is loaded.

logical-info output

INT .EXT:ref:*

if present and if *logical-info-maxlen* is not 0, points to a buffer that returns a set of logical attributes for the specified device. The attribute values are returned in a contiguous array.

If this parameter is present, *logical-info-maxlen* and *logical-info-len* must also be present.

For a description of the attributes returned in *logical-info*, see Device Attributes and Value Representations on page 4-55.

logical-info-maxlen input

INT:value

specifies the length in bytes of the buffer pointed to by *logical-info*. If the buffer length is too short for the full set of device attributes, the procedure returns as many values as will fit in the buffer.

This parameter must be present if *logical-info* is present.

logical-info-len output

INT .EXT:ref:1

returns the actual length in bytes of the buffer pointed to by *logical-info*.

This parameter must be present if *logical-info* is present.

primary-info output

INT .EXT:ref:*

if present and if *primary-info-maxlen* is not 0, points to a buffer that returns a set of physical device attributes obtained from the primary I/O process that supports the specified device. The attribute values are returned in a contiguous array.

If this parameter is present, *primary-info-maxlen* and *primary-info-len* must also be present.

For a description of the attributes returned in *primary-info*, see Device Attributes and Value Representations on page 4-55.

primary-info-maxlen input

INT:value

specifies the length in bytes of the buffer pointed to by *primary-info*. If the buffer length is too short for the full set of device attributes, the procedure returns as many values as will fit in the buffer.

This parameter must be present if *primary-info* is present.

primary-info-len output

INT .EXT:ref:1

returns the actual length in bytes of the buffer pointed to by *primary-info*.

This parameter must be present if *primary-info* is present.

backup-info output

INT .EXT:ref:*

if present and if *backup-info-maxlen* is not 0, points to a buffer that returns a set of physical device attributes obtained from the backup I/O process that supports the specified device. The attribute values are returned in a contiguous array.

If this parameter is present, *backup-info-maxlen* and *backup-info-len* must also be present.

The set of attributes for which values are returned in *backup-info* is identical to the set returned in *primary-info*. For a description of the attributes returned in *backup-info*, see Device Attributes and Value Representations on page 4-55.

backup-info-maxlen input

INT:value

specifies the length in bytes of the buffer pointed to by *backup-info*. If the buffer length is too short for the full set of device attributes, the procedure returns as many values as will fit in the buffer.

This parameter must be present if *backup-info* is present.

backup-info-len output

INT .EXT:ref:1

returns the actual length in bytes of the buffer pointed to by *backup-info*.

This parameter must be present if *backup-info* is present.

timeout input

INT(32):value

specifies how many hundredths of a second the procedure should wait for a response from the I/O process. The maximum value is 2147483647. The default

value is 6000D (one minute). A value of -1D causes the procedure to wait indefinitely.

options input

INT:value

specifies options. The bits, when set, indicate:

<0:12> Reserved (specify 0)

<13> Specifies that the procedure search for the next device that has a subtype of *match-subtype*. *options*.<15> must be equal to 1 and *match-subtype* must be specified when this option is used.

<14> Specifies that the procedure search for the next device that has a type of *match-type*. *options*.<15> must be equal to 1 and *match-type* must be specified when this option is used.

<15> Specifies that the procedure search for the next device that matches the selection criteria. The search begins with the logical device number immediately following the one specified by the *ldevnum* parameter. This option can be used alone or in combination with *options*.<13> or *options*.<14>. See “Considerations,” later in this subsection.

The default value of *options* is 0.

match-type input

INT:value

if present and if not -1, specifies a device type that is to be used as a search criterion. Supplying *match-type* causes the procedure to return information for the next device that has a device type of *match-type* and a logical device number greater than *ldevnum*.

options.<14> and *options*.<15> must both be equal to 1 in order to use this parameter.

match-type can also be used in combination with *match-subtype*. See “Considerations,” later in this subsection.

match-subtype input

INT:value

if present and if not -1, specifies a device subtype that is to be used as a search criterion. Supplying *match-subtype* causes the procedure to return information for the next device that has a device subtype of *match-subtype* and a logical device number greater than *ldevnum*.

options.<13> and *options*.<15> must both be equal to 1 in order to use this parameter.

match-subtype can also be used in combination with *match-type*. See "Considerations," later in this subsection.

devname:*maxlen* output: input

STRING .EXT:ref:*, INT:value

if supplied and if *maxlen* is not 0, returns a local name (that is, a name that does not include a node name) designating the device. The returned name has no qualifiers.

If the device does not have a name, a *devname-len* of 0 is returned. A *devname* in the form $*logical-device-number* is never returned.

maxlen specifies the length in bytes of the string variable *devname*.

devname-len output

INT .EXT:ref:1

returns the actual length in bytes of the name returned in *devname*. If the device does not have a name, a *devname-len* of 0 is returned.

This parameter must be present if *devname* is present.

error-detail output

INT .EXT:ref:1

for some returned errors, contains additional information. See *error*, earlier in this subsection.

Considerations

- I/O process status

  The physical information that is returned in *primary-info* and *backup-info* includes a status field (see Device Attributes and Value Representations on page 4-55). This field contains a file-system error number that indicates the result of the request for information from the I/O process.

  It is possible for DEVICE_GETINFOBYLDEV_ to return an *error* value of 0 (information successfully returned) while the IOP reports an error in the status field. In that case, the *error* value of 0 indicates that communication with the IOP was successful, while the IOP status value reflects the validity of the returned information.

- Searching logical devices

  To perform a search of logical devices, you must specify *options*.<15> = 1. DEVICE_GETINFOBYLDEV_ searches logical device numbers starting with the number immediately following *ldevnum*. Information is returned for the next device found.

To search all logical devices, set the initial value of *ldevnum* to -1; for each iteration of the search, update *ldevnum* to the logical device number of the last device for which information was returned. The logical device number of a device is returned in *logical-info* (see Device Attributes and Value Representations).

When *match-type* is supplied (in combination with *options*.<14> equal to 1), or when *match-subtype* is supplied (in combination with *options*.<13> equal to 1), the search returns information only for a device of the specified type or subtype. *match-type* and *match-subtype* can be used together; in that case, a device must match both the specified type and subtype to be selected by the search.

When a search can find no more devices, an *error* value of 4 is returned and *error-detail* contains 19 (no more devices).

Device Attributes and Value Representations

These set of attributes is returned in *logical-info* if that parameter is present:

Attribute	TAL Value Representation
ldev	INT(32)
primary-processor	INT
primary-PIN	INT
backup-processor	INT
backup-PIN	INT
type	INT
subtype	INT
record-size	INT
audited	UNSIGNED(1)
dynamically-configured	UNSIGNED(1)
demountable	UNSIGNED(1)
has-subnames (12 bits of filler)	UNSIGNED(1)

The attributes returned in *logical-info* are defined as follows:

- ldev

  is the logical device number of the device for which information has been obtained. On G-series RVUs, the logical device number of a device can change whenever a device is configured or the system is loaded.

- primary-processor

  is the number of the processor in which the primary IOP that owns the device is running.

- primary-PIN

is process identification number (PIN) of the primary IOP that owns the device.

- backup-processor

  is the number of the processor in which the backup IOP that owns the device is running.

- backup-PIN

  is the process identification number (PIN) of the backup IOP that owns the device.

- type

  is the device type of the device. See Appendix A, Device Types and Subtypes for a list of device types.

- subtype

  is the device subtype of the device. See Appendix A, Device Types and Subtypes for a list of device subtypes.

- record-size

  is the record size of the device.

- audited

  if equal to 1, indicates that the device is TMF audited.

- dynamically-configured

  if equal to 1, indicates that the device was configured dynamically instead of with SYSGEN.

- demountable

  if equal to 1, indicates that the device is logically demountable.

- has-subnames

  if equal to 1, indicates that the device has subdevices that can be opened (for example, $DEVICE.#SUBDEV).

These set of attributes is returned in *primary-info* and *backup-info* if those parameters are present:

Attribute	TAL Value Representation
status	INT
primary-subtype	INT
mirror-subtype	INT
has-physical-devices	UNSIGNED(1)
is-primary (14 bits of filler)	UNSIGNED(1)
path 0 information:	

Attribute	TAL Value Representation
configured	UNSIGNED(1)
in-use (14 bits of filler)	UNSIGNED(1)
channel	INT
controller	INT
unit	INT
state	INT
path 1 information:	
configured	UNSIGNED(1)
in-use (14 bits of filler)	UNSIGNED(1)
channel	INT
controller	INT
unit	INT
state	INT
path 2 information:	
configured	UNSIGNED(1)
in-use (14 bits of filler)	UNSIGNED(1)
channel	INT
controller	INT
unit	INT
state	INT
path 3 information:	
configured	UNSIGNED(1)
in-use (14 bits of filler)	UNSIGNED(1)
channel	INT
controller	INT
unit	INT
state	INT

The attributes returned in *primary-info* and *backup-info* are defined as follows:

- status

is a file-system error number returned by the IOP that owns the device. A value of 0 indicates that the returned information is valid; any other value indicates an error condition.

- primary-subtype

  is the device subtype of the primary disk of a logical volume. This field is set only by the disk process.

- mirror-subtype

  is the device subtype of the mirror disk of a logical volume. This field is set only by the disk process.

- has-physical-devices

  is equal to 1 unless the logical device does not own a channel address. $TMP, $0, and $IPB are examples of logical devices that do not own channel addresses.

- is-primary

  identifies the current primary process of the IOP pair. This bit should be set to 1 in only one of the physical information sets.

- configured

  is equal to 1 if the path is known to the device. Devices such as terminals and tape drives have only one path configured; disks can have two or four paths configured.

- in-use

  is equal to 1 if the path is currently in use by the IOP that owns the device.

- channel

  is the channel number of the path. -1 is always returned on G-series RVUs, indicating that this parameter value is not returned.

- controller

  is the controller number of the path. -1 is always returned on G-series RVUs, indicating that this parameter value is not returned.

- unit

  is the unit number of the path. -1 is always returned on G-series RVUs, indicating that this parameter value is not returned.

- state

  is the current state of the path. If the device has only one path, then the state of the device is the state of the path. Valid state values include:

Value	Description
0	UP
1	DOWN
2	SPECIAL
3	MOUNT
4	REVIVE
5	(reserved)
6	EXERCISE
7	EXCLUSIVE
8	HARD DOWN
9	UNKNOWN

Example

```
! obtain logical and physical attributes for
! logical device 10.

logical^device := 10;
error := DEVICE_GETINFOBYLDEV_ (
                 logical^device,
                 l^info, l^info^maxlen, l^info^len,
                 p^info, p^info^maxlen, p^info^len,
                 b^info, b^info^maxlen, b^info^len );
```

Related Programming Manual

For programming information about the DEVICE_GETINFOBYLDEV_ procedure, see the *Guardian Programmer's Guide.*

DEVICE_GETINFOBYNAME_ Procedure (Superseded on G-Series RVUs)

Summary

Note. On G-series RVUs, this procedure is supported for compatibility with previous software and should not be used for new development. This procedure cannot obtain all of the physical attributes of a device. For new development, call the CONFIG_GETINFO_BYNAME_ procedure.

The DEVICE_GETINFOBYNAME_ procedure obtains the physical and logical attributes of a device. The device is specified by name.

Syntax for C Programmers

```
#include <cextdecs(DEVICE_GETINFOBYNAME_)>

short DEVICE_GETINFOBYNAME_ ( char *devname
                             ,short length
                             ,[ short *logical-info ]
                             ,[ short logical-info-maxlen ]
                             ,[ short *logical-info-len ]
                             ,[ short *primary-info ]
                             ,[ short primary-info-maxlen ]
                             ,[ short *primary-info-len ]
                             ,[ short *backup-info ]
                             ,[ short backup-info-maxlen ]
                             ,[ short *backup-info-len ]
                             ,[ __int32_t timeout ]
                             ,[ short *error-detail ] );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
error := DEVICE_GETINFOBYNAME_ ( devname:length          !
i:i
                                ,[ logical-info ]        ! o
                                ,[ logical-info-maxlen ] ! i
                                ,[ logical-info-len ]    ! o
                                ,[ primary-info ]        ! o
                                ,[ primary-info-maxlen ] ! i
                                ,[ primary-info-len ]    ! o
                                ,[ backup-info ]         ! o
                                ,[ backup-info-maxlen ]  ! i
                                ,[ backup-info-len ]     ! o
                                ,[ timeout ]             ! i
                                ,[  error-detail ] );     ! o
```

Parameters

error returned value

INT

indicates the outcome of the operation. It returns one of these values:

0 Information was successfully returned

1 (reserved)

2 Parameter error; *error-detail* contains the number of the first parameter found to be in error, where 1 designates the first parameter on the left.

3 Bounds error; *error-detail* contains the number of the first parameter found to be in error, where 1 designates the first parameter on the left.

4 Device not found; *error-detail* contains a file-system error number.

devname:*length* input:input

STRING .EXT:ref:*, INT:value

specifies the name of the device for which information is requested. *devname* must be a local name (that is, it must not include a system name) and must have no qualifiers.

A *devname* in the form $*logical-device-number* (for example, $23) is acceptable.

devname must be exactly *length* bytes long.

logical-info output

INT .EXT:ref:*

if present and if *logical-info-maxlen* is not 0, returns a set of logical attributes for the specified device. The attribute values are returned in a contiguous array.

If this parameter is present, *logical-info-maxlen* and *logical-info-len* must also be present.

The set of attributes for which values are returned in *logical-info* is identical to the set returned in the *logical-info* parameter of DEVICE_GETINFOBYLDEV_. For a description of the attributes returned in *logical-info*, see Device Attributes and Value Representations on page 4-55.

logical-info-maxlen input

INT:value

specifies the length in bytes of the buffer pointed to by *logical-info*. If the buffer length is too short for the full set of device attributes, the procedure returns as many values as will fit in the buffer.

This parameter must be present if *logical-info* is present.

logical-info-len output

INT .EXT:ref:1

returns the actual length in bytes of the buffer pointed to by *logical-info*.

This parameter must be present if *logical-info* is present.

primary-info output

INT .EXT:ref:*

if present and if *primary-info-maxlen* is not 0, points to a buffer that returns a set of physical device attributes obtained from the primary I/O process that supports the specified device. The attribute values are returned in a contiguous array.

If this parameter is present, *primary-info-maxlen* and *primary-info-len* must also be present.

The set of attributes for which values are returned in *primary-info* is identical to the set returned in the *primary-info* parameter of DEVICE_GETINFOBYLDEV_. For a description of the attributes returned in *primary-info*, see Device Attributes and Value Representations on page 4-55.

primary-info-maxlen input

INT:value

specifies the length in bytes of the buffer pointed to by *primary-info*. If the buffer length is too short for the full set of device attributes, the procedure returns as many values as will fit in the buffer.

This parameter must be present if *primary-info* is present.

primary-info-len output

INT .EXT:ref:1

returns the actual length in bytes of the buffer pointed to by *primary-info*.

This parameter must be present if *primary-info* is present.

backup-info output

INT .EXT:ref:*

if present and if *backup-info-maxlen* is not 0, points to a buffer that returns a set of physical device attributes obtained from the backup I/O process that supports the specified device. The attribute values are returned in a contiguous array.

If this parameter is present, *backup-info-maxlen* and *backup-info-len* must also be present.

The set of attributes for which values are returned in *backup-info* is identical to the set returned in the *primary-info* parameter of this procedure and of the DEVICE_GETINFOBYLDEV_ procedure. For a description of the attributes returned in *backup-info*, see Device Attributes and Value Representations on page 4-55

backup-info-maxlen input

INT:value

specifies the length in bytes of the buffer pointed to by *backup-info*. If the buffer length is too short for the full set of device attributes, the procedure returns as many values as will fit in the buffer.

This parameter must be present if *backup-info* is present.

backup-info-len output

INT .EXT:ref:1

returns the actual length in bytes of the buffer pointed to by *backup-info*.

This parameter must be present if *backup-info* is present.

timeout input

INT(32):value

specifies how many hundredths of a second the procedure should wait for a response from the I/O process. The maximum value is 2147483647. The default value is 6000D (one minute). A value of -1D causes the procedure to wait indefinitely.

error-detail output

INT .EXT:ref:1

for some returned errors, contains additional information. See *error*, earlier in this subsection.

Considerations

- I/O process status

  The physical information that is returned in *primary-info* and *backup-info* includes a status field (see Device Attributes and Value Representations on page 4-55). This field contains a file-system error number that indicates the result of the request for information from the I/O process.

  It is possible for DEVICE_GETINFOBYNAME_ to return an *error* value of 0 (information successfully returned) while the IOP reports an error in the status field. In that case, the *error* value of 0 indicates that communication with the IOP was successful, while the IOP status value reflects the validity of the returned information.

Example

```
! obtain logical and physical information for
! the device named "$TERM11".

device^name ':=' "$TERM11";
error := DEVICE_GETINFOBYNAME_ (
                 device^name : 7,
                 l^info, l^info^maxlen, l^info^len,
                 p^info, p^info^maxlen, p^info^len,
                 b^info, b^info^maxlen, b^info^len,
                 8000D, error^detail );
```

Related Programming Manual

For programming information about the DEVICE_GETINFOBYNAME_ procedure, see the *Guardian Programmer's Guide*.

DEVICEINFO Procedure (Superseded by FILE_GETINFOBYNAME_ Procedure or FILE_GETINFOLISTBYNAME_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development. DEVICEINFO cannot obtain information on devices that have a device type greater than 63.

The DEVICEINFO procedure is used to obtain the device type and the physical record length of a file. The file can be opened or closed.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
CALL DEVICEINFO ( file-name                    ! i
                 ,devtype                      ! o
                 ,physical-recordlen );        ! o
```

Parameters

file-name input

INT:ref:12

is an array containing the name of the device whose characteristics are to be returned. Any form of 12-word internal-format file name is permitted. For disk files, only the first eight characters (that is, the volume name) are significant; however, the remaining 16 characters still must be in a valid file name format. If a logical device number is specified, the last 16 characters must be blanks.

devtype output

INT:ref:1

returns the device type of the associated file in this form:

<0>	Demountable
<1>	Audited disk, or the file name specified was a subdevice
<2:3>	Undefined
<4:9>	Device type
<10:15>	Device subtype

If the device type is greater than 63, bits <4:9> are set to 44. To obtain information on devices with a device type greater than 63, call either the FILE_GETINFOBYNAME_ or FILE_GETINFOLISTBYNAME_ procedure. For a list of the device types, see Appendix A, Device Types and Subtypes.

physical-recordlen output

INT:ref:1

returns the physical record length associated with the file. Physical record length is determined as follows:

nondisk devices
physical-recordlen is the configured record length.

disk files
physical-recordlen is the maximum possible transfer length. The transfer length is equal to the configured buffer size for the device (either 2048 or 4096 bytes). (For an Enscribe disk file, the logical record length can be obtained through the FILE_GETINFO[BYNAME]_ , FILE_GETINFOLIST[BYNAME]_ , or FILERECINFO procedure.)

processes and $RECEIVE file
a length of 132 is returned in *physical-recordlen*. This is the system convention for interprocess files.

Considerations

- All parameters are required. If any are missing, or in error, DEVICEINFO returns with no error indication.
- When DEVICEINFO is called with a file name that designates a subtype 30 process, it sends a device-type inquiry system message to the process to determine the device type and subtype. The format of this completion message is described in the *Guardian Procedure Errors and Messages Manual*.
- A deadlock occurs if a subtype 30 process calls DEVICEINFO on its own process name.
- For interprocess messages directed to a process pair, file-system errors 200 (ownership) and 201 (path down) are retried automatically to the other member of the pair.

DEVICEINFO2 Procedure (Superseded by FILE_GETINFOBYNAME_ Procedure or FILE_GETINFOLISTBYNAME_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development. DEVICEINFO2 cannot obtain information on devices that have a device type greater than 63.

The DEVICEINFO2 procedure is used to obtain the device type and the physical record length of a file, which can be open or closed.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
CALL DEVICEINFO2 ( file-name                      ! i
                 , [ devtype ]                    ! o
                 , [ physical-recordlen ]         ! o
                 , [ diskprocess-version ]        ! o
                 , [ error ]                      ! o
                 , [ options ]                    ! i
                 , [ tag-or-timeout ] );          ! i
```

Parameters

file-name input

INT:ref:12

is an array containing the name of the device whose characteristics are to be returned. Any form of 12-word internal format file name is permitted. For disk files, only the first eight characters (that is, the volume name) are significant; however,

the remaining 16 characters still must be in a valid file name format. If a logical device number is specified, the last 16 characters must be blanks.

devtype output

INT:ref:1

returns the device type of the associated file in this form:

<0>	Demountable
<1>	Audited disk, or file name specified was a subdevice
<2:3>	Undefined
<4:9>	Device type
<10:15>	Device subtype

If the device type is greater than 63, bits <4:9> are set to 44. To obtain information on devices with a device type greater than 63, call either the FILE_GETINFOBYNAME_ or FILE_GETINFOLISTBYNAME_ procedure. For a list of the device types, see Appendix A, Device Types and Subtypes.

physical-recordlen output

INT:ref:1

returns the physical record length associated with the file:

nondisk devices
physical-recordlen is the configured record length.

disk files
physical-recordlen is the maximum possible transfer length. Transfer length is equal to the configured buffer size for the device (either 2048 or 4096 bytes). (For an Enscribe disk file, the logical record length can be obtained through the FILE_GETINFO[BYNAME]_ , FILE_GETINFOLIST[BYNAME]_ , or FILERECINFO procedure.)

processes and $RECEIVE file
a length of 132 is returned in *physical-recordlen*. This is the system convention for interprocess files.

diskprocess-version output

INT:ref:1

returns the disk process version for disk devices.

(*devtype*.<4:9> = 3):

diskprocess-version 1 DP2 disk process.

error output

INT:ref:1

is a file-system error number indicating the success of the call or the reason for its failure.

options input

INT:value

is a word indicating the options desired:

<0:12>
should be zero.

<13> if 1, indicates that this call is initiating a nowait inquiry and the information will be returned in a system message. Do not set both *options*.<13> and *options*.<14> to 1. See "Considerations" for more information.

<14> if 1, indicates that the sending of device-type inquiry messages to a subtype 30 process should not be allowed to take longer than indicated by the timeout value in *tag-or-timeout*. If the time is exceeded, error 40 is returned.

<15> if 1, indicates that device-type inquiry messages are not to be sent to subtype 30 processes.

If omitted, zero is used.

tag-or-timeout input

INT(32):value

is a parameter with two functions depending on the *options* you specify:

- If *options*.<13> = 1, it is a value you define that helps identify one of several DEVICEINFO2 operations. The system stores this value until the operation completes, then returns it to the program in words 1 and 2 of a system message. See "Considerations" for more information.
- If *options*.<14> = 1, it is the maximum amount of time, in 0.01-second units, to wait. The value -1D indicates an indefinite wait. If this parameter is omitted, -1D is used.

Considerations

- When DEVICEINFO2 is called with a file name that designates a subtype 30 process, it sends a device-type inquiry system message to the process to determine the device type and subtype (unless disabled by the *options* parameter). The format of this completion message is described in the *Guardian Procedure Errors and Messages Manual*.

  A deadlock occurs if a subtype 30 process calls DEVICEINFO on its own process name.

- If you call this procedure in a nowait manner (*options* <13> is set to 1), the results are returned in the nowait DEVICEINFO2 completion message (-41), not in the output parameters of the procedure. The format of this completion message is described in the *Guardian Procedure Errors and Messages Manual*. If *error* is not 0, no completion message is sent to $RECEIVE. Errors can be reported either on return from the procedure, in which case *error* might be meaningful, or through the completion message sent to $RECEIVE.

  The system reports a path error only after automatically making retries.

  The nowait option allows any step of the inquiry process to execute asynchronously to the caller. However, this option guarantees only that simulation inquiries to subtype 30 processes will be asynchronous. Other parts of the function may or may not be asynchronous.

  Process pairs using the nowait option should handle the fact that a DEVICEINFO2 completion message is delivered only to the process that initiates it, not to the other member of the pair. You might have the primary process keep the backup process ignorant of outstanding inquiries, or you might have equivalent DEVICEINFO2 calls at the point where a backup takes over from the primary process.

  For interprocess messages directed to a process pair, file-system errors 200 (ownership) and 201 (path down) are retried automatically to the other member of the pair.

  Switching ownership from the primary to the backup process can leave outstanding inquiries. The CHECKSWITCH procedure automatically discards these as it becomes the backup process. Programs using another method of switching should tolerate the completions of these irrelevant inquiries.

Example

```
CALL DEVICEINFO2 ( INFILE, DEVTYPE, RECLENGTH, D^VERSION );
```

DISK_REFRESH_ Procedure

Summary

Note. On G-series RVUs, this procedure is supported for compatibility with previous software and should not be used for new development; the function that it provides is no longer needed.

The DISK_REFRESH_ procedure causes control information to be written to the specified disk volume. DISK_REFRESH_ always writes out the control information contained in file control blocks (FCBs), such as end-of-file (EOF) pointers. Only the data and control information that is not already on disk is written.

The DISK_REFRESH_ procedure also writes all dirty (that is, modified) cache blocks to disk. The writing of cache blocks takes priority over all other disk activity and can severely affect response time on the disk volume. For this reason, the DISK_REFRESH_ procedure should not be used when performance of other programs is critical.

On RVUs preceding G00, The DISK_REFRESH_ procedure can be used when a volume is brought down (for example, immediately before a system load or PUP DOWN ! command) but should not be used at other times. On these RVUs, the DISK_REFRESH_ procedure or the equivalent Peripheral Utility Program (PUP) REFRESH command should be performed on all volumes before a total system shutdown.

On G-series RVUs, the DISK_REFRESH_ procedure is not needed because the system performs the equivalent operation automatically for each disk volume when it is brought down and at system shutdown.

Syntax for C Programmers

```
#include <cextdecs(DISK_REFRESH_)>

short DISK_REFRESH_ ( char *name
                     ,short length );
```

Syntax for TAL Programmers

```
error := DISK_REFRESH_ ( name:length );   ! i:i
```

Parameters

error returned value

INT

is a file-system error number indicating the outcome of the operation.

name:*length* input:input

STRING .EXT:ref:*, INT:value

specifies the name of the disk volume (or a file on the disk volume) that is to have control information written out. If a disk file is specified, the operation is performed for the entire volume on which the file resides.

The value of *name* must be exactly *length* bytes long and must be a valid file (or volume) name or DEFINE name. If the name is partially qualified, it is resolved using the contents of the VOLUME attribute of the =_DEFAULTS DEFINE.

Considerations

- Because calling the DISK_REFRESH_ procedure can severely impact response time on the specified disk volume, these actions might be considered as alternatives:
  - When creating a file using FILE_CREATE_ , FILE_CREATELIST_ , or CREATE, select the option that causes the file label to be written immediately to disk whenever the EOF value changes.
  - Use SETMODE function 95 to cause the dirty cache buffers of a specified file to be written to disk.

Example

```
error := DISK_REFRESH_ ( volume^name:length );
```

DISKINFO Procedure (Superseded by FILE_GETINFOLISTBYNAME_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

This procedure obtains information about disk volumes.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
error := DISKINFO ( name                          ! i
                  ,[ capacity ]                   ! o
                  ,[ avail ]                      ! o
                  ,[ numfrag ]                    ! o
                  ,[ biggest ]                    ! o
                  ,[ drivekinds ]                 ! o
                  ,[ drivecaps ] );               ! o
```

Parameters

error returned value

INT

is a file-system error number indicating the success of the call or the reason for its failure.

name input

INT .EXT:ref:12

is the name of the disk volume being inquired about. The name can be that of a disk volume (blank padded), or of a disk file, or of a DEFINE designating a disk volume or file. The name can refer to a disk on a different network node.

capacity output

INT(32) .EXT:ref:1

is the information capacity of the volume as labeled, in pages (2048 byte units). This value accounts for the space taken up for data protection (spare sectors, and so on), but does not account for space used by the operating system for volume management (such as directory space) or other uses.

avail output

INT(32) .EXT:ref:1

is the total free space currently available, in pages (2048 byte units).

numfrag output

INT(32) .EXT:ref:1

is the number of individual free space fragments.

biggest output

INT(32) .EXT:ref:1

is the size, in pages, of the largest free space fragment.

drivekinds output

STRING .EXT:ref:16

identifies the kinds of drives on which the volume is mounted. The value contains two fields of ASCII data:

`[0:7]` The product number of the primary drive

`[8:15]` The product number of the mirror drive

If information is unavailable for a drive (because it is inaccessible or not configured), its corresponding field will be blank. Drive models 4110 and 4120, which cannot be distinguished by software, are returned as “4110”. Similarly, drive model 4106 is returned as “4105”, 4111 as “4110”, and 4115 as “4114”.

The last four characters of the product-number fields sometimes contain either blanks or modifiers used to distinguish between different versions of a product. Also, for the 4105 model, an “M” or “F” modifier refers to the moving-head part or the fixed-head part of the drive.

drivecaps output

INT(32) .EXT:ref:2

are the formatted capacities of the primary and mirror drives, in pages. These values account for the space taken up for data protection (such as spare sectors), but not for other uses. If the information is unavailable for a drive (because it is inaccessible or not configured), its corresponding value is zero. These values are provided for cases in which different model drives are mirrored and thus must be labeled with the smaller of the two formatted capacities.

Consideration

You should always supply at least one output parameter when calling DISKINFO. If you supply no output parameters, the returned *error* value might vary with different device types of *name* and with different versions of the operating system.

Example

```
NAME ':=' "   " & NAME FOR 11;
NAME ':=' "$SYSTEM";
ERR := DISKINFO (NAME ,, FREE ,, BIG );
IF ERR <> 0 THEN ... !handle error
```

DNUMIN Procedure

Summary

The DNUMIN procedure converts the ASCII characters used to represent a number into the signed double word integer value for that number.

Syntax for C Programmers

Note. In the TNS/E environment the CEXTDECS file uses the *int* data type for 32-bit values. This is a change from the TNS and TNS/R environments where CEXTDECS uses the *long* data type for 32-bit values.

```
#include <cextdecs(DNUMIN)>

__int32_t DNUMIN ( char *ascii-num
                 ,__int32_t *signed-result
                   ,short base
                  ,[ short *status ]
                ,[ short flags ] );
```

Syntax for TAL Programmers

```
next-addr := DNUMIN ( ascii-num                ! i
                     ,signed-result            ! o
                     ,base                     ! i
                     ,[ status ]               ! o
                     ,[ flags ] );             ! i
```

Parameters

next-addr returned value

EXTADDR

returns the ‘G’[0] relative byte address of the first character in *ascii-num* after the number, or the last character examined in case of an error.

ascii-number input

STRING .EXT:ref:*

is an array containing the number to be converted to signed double word integer form. *ascii-number* is of the form:

```
[ + ] [ prefix ] number nonnumeric
[ - ]
```

where any of these *prefix* values can be used to override the specified *base*:

%	octal
#	decimal
%b or %B	binary
%h or %H	hexadecimal

The *number* cannot contain embedded commas.

signed-result output

INT(32) .EXT:ref:1

returns the signed double word integer result of the conversion.

base input

INT:value

specifies the number base of *ascii-number*. Legitimate values are 2 through 36. Note that supplying a *prefix* in *ascii-number* overrides this specification.

status output

INT .EXT:ref:1

returns a number that indicates the outcome of the conversion.

The values for *status* are:

1 Nonexistent number (string does not start with a valid sign, prefix, or numeric)

0 Valid conversion

-1 Invalid integer (number cannot be represented in 32 bits as a signed quantity)

flags input

INT:value

can be used to alter the number format accepted by DNUMIN as follows:

<0:12>	Must be 0
<13>	Disallow preceding sign (+/-)
<14>	Disallow prefixes (%, #, and so on)
<15>	Permit two-word *number* of the form *integer1.integer2* where each unsigned integer must fit in a 16-bit word.

If not supplied, *flags* defaults to zero.

Considerations

- Number conversion stops on the first ASCII character that either does not represent any numeric value or that represents a numeric value greater than (*base* -1). For bases greater than 10, where letters are used to represent the values 10 and above, both uppercase and lowercase letters are accepted. Embedded commas are not allowed.
- Decimal numeric values must be in the range -2,147,483,648 to +2,147,483,647.
- Numeric values in other number bases must be in the range -%h80000000 to +%hFFFFFFFF. DNUMIN accepts positive numbers in the 32-bit range and negative numbers in the 31-bit range.

Example

```
STRING .number[0:15] := ["-2147483640 "];
INT(32) result;
```

```
INT base := 10;
INT status := 0;
      .
      .
CALL DNUMIN ( number, result, base, status );
IF status <> 0 THEN ...
```

Related Programming Manual

For related programming information about the DNUMIN utility procedure, see the *Guardian Programmer's Guide*.

DNUMOUT Procedure

Summary

The DNUMOUT procedure converts unsigned double word integer values to their ASCII equivalents. The result is returned right-justified in an array. If necessary, leading zeros are zero-filled (the default) or blank-filled.

Syntax for C Programmers

```
#include <cextdecs(DNUMOUT)>

short DNUMOUT ( char *ascii-result
                ,__int32_t unsigned-doubleword
                ,short base
                ,[ short width ]
                ,[ short flags ] );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
width := DNUMOUT ( ascii-result                ! o
                  ,unsigned-doubleword         ! i
                  ,base                        ! i
                  ,[ width ]                   ! i
                  ,[ flags ] );                ! i
```

Parameters

width returned value

INT

returns the length in bytes of the *ascii-result*.

ascii-result output

STRING .EXT:ref:*

is an array where the converted value is returned in ASCII representation, right-justified in *ascii-result*[*width* - 1], with zero-fill by default.

unsigned-doubleword input

INT(32):value

is the unsigned double-word integer to be converted.

base input

INT:value

is the number base for the resulting conversion. Any number in the range 2 through 36 is valid.

width input

INT:value

is the maximum number of characters permitted in *ascii-result*. The output value will be right-justified to the specified width; characters may be truncated on the left side. If *width* is negative or not supplied, DNUMOUT calculates and uses the minimum width necessary to hold the entire value.

flags input

INT:value

may be used to alter the formatting used by DNUMOUT as follows:

<0:14>
Must be zero

<15>	1	Blank-fill on left
	0	Zero-fill on left (the default)

If not supplied, *flags* defaults to zero.

Considerations

If *width* is too small to contain the number, the most significant digits are lost.

Example

```
STRING .buffer[0:10] := ["           "];
INT(32) dnum := 2147483640D;
INT base := 10;
INT width := -1;
      .
      .
CALL DNUMOUT ( buffer, dnum, base, width );
IF width < 0 THEN ...
```

Related Programming Manual

For related programming information about the DNUMOUT utility procedure, see the *Guardian Programmer's Guide*.

DST_GETINFO_ Procedure

Summary

The DST_GETINFO_ procedure, for G05.00 and later G-series RVUs, provides the information about the DST entry that is in effect at time *keygmt*.

Syntax for C Programmers

```
#include <cextdecs(DST_GETINFO_)>

short DST_GETINFO_ ( long long keygmt
                    ,short *dstentry );
```

Syntax for TAL Programmers

```
error:= DST_GETINFO_  (  keygmt              ! i
                       , dstentry );         ! o
```

Parameters

error

indicates the outcome of the operation. Table 4-2, Error Summary for DST_* Procedures, summarizes the possible values for *error*. It is recommended that the literal values be used instead of the numeric values when coding (for example, ZSYS^VAL^DST^OK, not 0).

keygmt input

FIXED:value

specifies time in GMT for required DST information. If *keygmt* is set to 0, it returns the entry currently in effect. If *keygmt* is set to -1, it returns the first entry in the table. If *keygmt* is set to 1, it returns the last entry in the table.

dstentry output

INT.EXT:ref:*

specifies the address of the ZSYS^DDL^DST^ENTRY^DEF structure that will be filled in with the required information.

Considerations

- If the *keygmt* value is less the than the *lowgmt* value of the first entry in the table with nonzero offset, ZSYS^VAL^DST^RANGE^LOW (error 9) is returned.
- If the *keygmt* value is greater than or equal to the *highgmt* value of the last entry in the table with nonzero offset, ZSYS^VAL^DST^RANGE^HIGH (error 10) is returned.

Example

```
#include <cextdecs (DST_GETINFO_)>
short error;
long long keyGMT;
zsys_ddl_dst_entry_def dstentry;

dstentry.z_version = ZSYS_VAL_DST_VERSION_SEP1997;
error = DST_GETINFO_ (keyGMT, (short*)&dstentry);
```

DST_TRANSITION_ADD_ Procedure

Summary

The DST_TRANSITION_ADD_ procedure, for G05.00 and later G-series RVUs, allows a super-group user (255,*n*) to add an entry to the daylight-saving-time (DST) transition table. This operation is allowed only when the DAYLIGHT_SAVING_TIME option in the system is configured to the TABLE option.

Syntax for C Programmers

```
#include <cextdecs(DST_TRANSITION_ADD_)>

short DST_TRANSITION_ADD_ ( short *dstentry );
```

Syntax for TAL Programmers

```
error:= DST_TRANSITION_ADD_ ( dstentry );                    ! i
```

Parameters

error

indicates the outcome of the operation. Table 4-2, Error Summary for DST_* Procedures, summarizes the possible values for *error*. It is recommended that

the literal values be used instead of the numeric values when coding (for example, ZSYS^VAL^DST^OK, not 0).

dstentry input

INT.EXT:ref:*

specifies the address of the ZSYS^DDL^DST^ENTRY^DEF structure that contains all of the input fields for this procedure. For more information on how to assign field values to the structure, see Structure Definition for dstentry on page 4-84.

Table 4-2. Error Summary for DST_* Procedures

Error	Literal Value	Description
0	ZSYS^VAL^DST^OK	The operation is successful.
1	ZSYS^VAL^DST^SECURITY^ERROR	The caller is not a super-group user (255,*n*).
2	ZSYS^VAL^DST^BAD^VERSION	The version number passed in ZSYS^DDL^DST^ENTRY^DEF is not valid. The only valid version is ZSYS^VAL^DST^VERSION^SEP1997.
3	ZSYS^VAL^DST^BAD^PARAMETER	One of the specified parameters is not valid.
4	ZSYS^VAL^DST^INTERVAL^ERROR	Invalid interval operation. An attempt was made to add, delete, or modify a DST entry that causes a collision with an existing DST entry.
5	ZSYS^VAL^DST^DELETE^NOW^ERROR	An attempt was made to delete a required DST entry. This error is returned when the DST entry that the user attempted to delete is in effect at the time the delete operation was attempted and the offset of the entry is nonzero.
6	ZSYS^VAL^DST^TYPE^ERROR	The DAYLIGHT_SAVING_TIME option in the system is not configured to use the TABLE option.
7	ZSYS^VAL^DST^TABLE^EMPTY	The DST table has no entries. This error is returned by the DST_GETINFO_ procedure.

Table 4-2. Error Summary for DST_* Procedures

Error	Literal Value	Description
8	ZSYS^VAL^DST^BOUNDS^ERROR	An attempt was made to use time values outside the supported range. The supported range is 1/ 1/ 1 0:00:00.000000 through 10000/12/31 23:59:59.999999 GMT.
9	ZSYS^VAL^DST^RANGE^LOW	The specified *keygmt* value was less than the *lowgmt* value of the first DST interval with nonzero offset. This error is returned by the DST_GETINFO_ procedure.
10	ZSYS^VAL^DST^RANGE^HIGH	The specified *keygmt* value is greater than the *highgmt* of the last DST interval with nonzero offset. This error is returned by the DST_GETINFO_ procedure.
11	ZSYS^VAL^DST^COUNT^OVERFLOW	An attempt was made to add too many entries to the table. Delete some of the entries and try again.

Structure Definition for *dstentry*

The *dstentry* parameter specifies the attributes of the new process.

In the TAL ZSYSTAL file, the structure for the *dstentry* parameter is defined as:

```
STRUCT ZSYS^DDL^DST^ENTRY^DEF (*)
?IF PTAL
FIELDALIGN (SHARED2)
?ENDIF PTAL

   BEGIN
   FIXED       Z^LOWGMT;
   FIXED       Z^HIGHGMT;
   INT         Z^OFFSET;
   INT         Z^VERSION;
   INT(32)     Z^FILLER;
   END;
```

Z^LOWGMT

identifies the lower limit of the interval in Greenwich Mean Time (GMT).

Z^HIGHGMT

identifies the higher limit of the interval in GMT.

Z^OFFSET

identifies the offset value of a transition.

Z^VERSION

identifies the version of the ZSYS^DDL^DST^ENTRY^DEF structure.

Z^FILLER

is provided for future use.

Considerations

- All time intervals that do not have explicit nonzero offset transition added are assumed to have a zero offset. Furthermore, all intervals that have a zero offset transition do not need to be explicitly added.
- All intervals that have nonzero offset transition must be explicitly added.
- Transitions can be added only if the interval that is to be added is completely covered by a zero offset interval in the table.

Example

```
#include <cextdecs (DST_TRANSITION_ADD_)>

zsys_ddl_dst_entry_def dstentry;
short error;
long long timeStampLow, timeStampHigh;

dstentry.z_lowgmt = timeStampLow;
dstentry.z_highgmt = timeStampHigh;
dstentry.z_offset = 3600; /* seconds */
dstentry.z_version = ZSYS_VAL_DST_VERSION_SEP1997;
error = DST_TRANSITION_ADD_ ((short*)&dstentry);
```

DST_TRANSITION_DELETE_ Procedure

Summary

The DST_TRANSITION_DELETE_ procedure, for G05.00 and later G-series RVUs, allows a super-group user (255,*n*) to delete an existing entry from the daylight saving time (DST) transition table. This operation is allowed only when the DAYLIGHT_SAVING_TIME option in the system is configured to the TABLE option.

Syntax for C Programmers

```
#include <cextdecs(DST_TRANSITION_DELETE_)>

short DST_TRANSITION_DELETE_ ( short *dstentry );
```

Syntax for TAL Programmers

```
error:= DST_TRANSITION_DELETE_ (  dstentry );                !
i
```

Parameters

error

indicates the outcome of the operation. Table 4-2, Error Summary for DST_* Procedures, summarizes the possible values for *error*. It is recommended that the literal values be used instead of the numeric values when coding (for example, ZSYS^VAL^DST^OK, not 0).

dstentry input

INT.EXT:ref:*

specifies the address of the ZSYS^DDL^DSTENTRY^DEF structure that contains all of the input fields for this procedure. For more information on how to assign field values to the structure, see Structure Definition for dstentry on page 4-84.

Considerations

- Only transition entries that already exist can be deleted. If an interval with nonzero offset covers the time at which the delete operation is attempted, ZSYS^VAL^DST^DELETE^NOW^ERROR (error 5) is returned.

Example

```
#include <cextdecs (DST_TRANSITION_DELETE_)>

zsys_ddl_dst_entry_def dstentry;
short error;
long long timeStampLow, timeStampHigh;

dstentry.z_lowgmt = timeStampLow;
dstentry.z_highgmt = timeStampHigh;
dstentry.z_offset = 3600; /* seconds */
dstentry.z_version = DST_VERSION_SEP1997;
error = DST_TRANSITION_DELETE_ ((short*)&dstentry);
```

DST_TRANSITION_MODIFY_ Procedure

Summary

The DST_TRANSITION_MODIFY_ procedure, for G05.00 and later G-series RVUs, allows a super-group user (255,*n*) to modify an entry in the daylight-saving-time (DST) transition table. This operation is allowed only when the DAYLIGHT_SAVING_TIME option in the system is configured to the TABLE option.

Syntax for C Programmers

```
#include <cextdecs(DST_TRANSITION_MODIFY_)>

short DST_TRANSITION_MODIFY_ ( short *olddst
                              ,short *newdst );
```

Syntax for TAL Programmers

```
error:= DST_TRANSITION_MODIFY_ (  olddst            ! i
                                , newdst );         ! i
```

Parameters

error

indicates the outcome of the operation. Table 4-2, Error Summary for DST_* Procedures, summarizes the possible values for *error*. It is recommended that the literal values be used instead of the numeric values when coding (for example, ZSYS^VAL^DST^OK, not 0).

olddst input

INT.EXT

specifies the address of an existing entry with a nonzero offset.

newdst input

INT.EXT

specifies the address of a new entry that is to take the place of *olddst*.

Considerations

- Transitions with nonzero offsets that already exist can be modified if the new values do not overlap other transitions with nonzero offsets that also exist.
- If you specify an offset value of zero for *newdst*, the *olddst* entry is deleted.

Example

```
#include <cextdecs (DST_TRANSITION_MODIFY_)>

zsys_ddl_dst_entry_def olddstentry, newdstentry;
short error;
long long oldTimeStampLow, oldTimeStampHigh;
long long newTimeStampLow, newTimeStampHigh;

olddstentry.z_lowgmt = oldTimeStampLow;
olddstentry.z_highgmt = oldTimeStampHigh;
olddstentry.z_offset = 3600; /* seconds */
olddstentry.z_version = ZSYS_VAL_DST_VERSION_SEP1997;
newdstentry.z_lowgmt = newTimeStampLow;
newdstentry.z_highgmt = newTimeStampHigh;
newdstentry.z_offset = 3600; /* seconds */
newdstentry.z_version = ZSYS_VAL_DST_VERSION_SEP1997;
error = DST_TRANSITION_MODIFY_ ((short*)&olddstentry,
(short*)&newdstentry);
```

EDITREAD Procedure

Summary

The EDITREAD procedure reads text lines from an EDIT file (file code = 101).

Text lines are transferred, in ascending order, from the text file to a buffer in the application program's data area. One line is transferred by each call to EDITREAD.

EDITREAD also returns the sequence number associated with the text line and performs checks to ensure that the text file is valid.

The EDIT file can be opened nowait. However, a call to EDITREAD completes before returning to the application program; it is not completed with a call to AWAITIO.

Note. Before EDITREAD is called, a call to EDITREADINIT must complete successfully.

Syntax for C Programmers

```
#include <cextdecs(EDITREAD)>

short EDITREAD ( short _near *edit-controlblk
                ,char *buffer
                ,short bufferlen
                ,__int32_t *sequence-num );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
status := EDITREAD ( edit-controlblk        ! i,o
                    ,buffer                 ! o
                    ,bufferlen              ! i
                    ,sequence-num );        ! o
```

Parameters

status returned value

INT

is a value indicating the outcome of EDITREAD. Values for *status* are:

>= 0 Indicates that the reading of the file was successful. This is the actual number of characters in the text line. However, no more than *bufferlen* bytes are transferred into *buffer*.

< 0 Indicates an unrecoverable error, where:

-1 End of file encountered

-2 Error occurred while reading

-3 Text file format error

-4 Sequence error. The sequence number of the line just read is less than its predecessor.

-5 Checksum error. EDITREADINIT was not called or the user has altered the edit control block. (This will not be returned if processing a reposition.)

-6 Invalid buffer address. Edit control block was not within lower half of user data stack.

edit-controlblk input, output

INT:ref:*

is an uninitialized array that is declared globally. The length in words of the edit control block must be at least 40 plus (*bufferlen* / 2). This control block is returned by EDITREADINIT and should be used in each call to EDITREAD. Do not modify it between calls.

buffer output

STRING:ref:*

is an array where the text line is to be transferred.

bufferlen input

INT:value

is the length, in bytes, of the *buffer* array. This specifies the maximum number of characters in the text line that is transferred into *buffer*.

sequence-num output

INT(32):ref:1

returns the sequence number multiplied by 1000, in double-word integer form, of the text line just read.

Example

```
COUNT := EDITREAD (CONTROL^BLOCK , LINE , LENGTH , SEQ^NUM);
```

If reading the file is successful, a count of the number of bytes in the text line returns in COUNT, the text line returns in the array LINE, and the sequence number returns in SEQ^NUM.

EDITREADINIT Procedure

Summary

The EDITREADINIT procedure is called to prepare a buffer in the application program's data area for subsequent calls to EDITREAD.

The application program designates an array to be used as an edit control block. The edit control block is used by the EDITREAD procedure for storing control information and as an internal buffer area.

The EDIT file can be opened nowait. However, a subsequent call to EDITREADINIT completes before returning to the application; it is not completed with a call to AWAITIO.

Syntax for C Programmers

```
#include <cextdecs(EDITREADINIT)>

short EDITREADINIT ( short _near *edit-controlblk
                    ,short filenum
                    ,short bufferlen );
```

Syntax for TAL Programmers

```
status := EDITREADINIT ( edit-controlblk        ! o
                        ,filenum                ! i
                        ,bufferlen );           ! i
```

Parameters

status returned value

INT

is a value indicating the outcome of EDITREADINIT. Values returned into *status* are:

0 Successful (OK to read)

-1 End of file detected (empty file)

-2 I/O error

-3 Format error (not EDIT file), or buffer length is incorrect

-6 Invalid buffer address. Edit control block was not within lower half of user data stack.

edit-controlblk output

INT:ref:*

is an uninitialized array that is declared globally. Forty words of the edit control block are used for control information. The remainder is used as an internal buffer by EDITREAD. The length, in words, of the edit control block must be at least 40 plus *bufferlen* divided by 2. This is the same array as specified in the *edit-controlblk* parameter to EDITREAD. You should not modify the contents of *edit-controlblk*.

filenum input

INT:value

is the number of an open file that identifies the text file to be read.

bufferlen input

INT:value

is the size, in bytes, of the internal buffer area used by EDITREAD. This parameter determines the amount of data that EDITREAD reads from the text file on disk (not the amount of data transferred into the buffer specified as a parameter to EDITREAD). The size of the internal buffer area must be a power of two, from 64 to 2048 bytes (that is, 64, 128, 256, ..., 2048).

Example

```
STAT := EDITREADINIT ( CONT^BLOCK , FNUM , BUF^LEN );
```

ERRNO_GET_ Procedure

Summary

The ERRNO_GET_ procedure obtains the value of the `errno` variable set by many OSS, native C/C++, and some Guardian routines.

Considerations for C Programmers

C programmers using the C run-time libraries, CRE support libraries, or shared run-time libraries (SRLs) can access the `errno` variable directly and do not use this procedure.

Syntax for TAL Programmers

```
?SOURCE $SYSTEM.SYSTEM.HERRNO

error := ERRNO_GET_;
```

Parameters

error returned value

INT(32)

returns the value of the `errno` variable.

Example

```
err := ERRNO_GET_;
```

Considerations

This procedure must be used to examine the `errno` value set by the SIGACTION_INIT_, SIGACTION_RESTORE_, SIGACTION_SUPPLANT_, and SIGJMP_SETMASK_ procedures when an error is detected.

EXTENDEDIT Procedure

Summary

The EXTENDEDIT procedure copies an EDIT file to a new file that it creates and that has a larger extent size than the original file. It purges the old file and renames the new file to have the name of the old file. The lines in the new file are renumbered if so requested. Upon completion, the current record number is set to -1 (beginning of file) and the file number of the new file is returned to the caller. This procedure is intended to be used after a call to WRITEEDIT or WRITEEDITP returns an error 45 (file full).

EXTENDEDIT is an IOEdit procedure and can only be used with files that have been opened by OPENEDIT or OPENEDIT_. The maximum edit file size is 128 megabytes.

Syntax for C Programmers

```
#include <cextdecs(EXTENDEDIT)>

__int32_t EXTENDEDIT ( short *filenum
                      ,[ __int32_t start ]
                      ,[ __int32_t increment ] );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
error := EXTENDEDIT ( filenum                    ! i,o
                     ,[ start ]                  ! i
                     ,[ increment ] );           ! i
```

Parameters

error returned value

INT

returns a file-system error number indicating the outcome of the operation. The most common cause of failure is error 48 (security violation), which occurs when the caller is not authorized to rename or purge the existing file.

filenum input, output

INT .EXT:ref:1

specifies the file number of the open file to be copied into a new file. It returns the file number of the new file.

start input

INT(32):value

specifies 1000 times the line number of the first line of the new file. You supply this parameter when you want the lines in the new file to be renumbered. If you omit *start*, renumbering still occurs if *increment* is present, in which case the value of *increment* is used for *start*. The possible EDIT line numbers are 0, 0.001, 0.002, ... 99999.999.

increment input

INT(32):value

if present and greater than 0, causes EXTENDEDIT to renumber the lines in the new file using the incremental value specified. The possible EDIT line numbers are 0, 0.001, 0.002, ... 99999.999. The value of *increment* indicates 1000 times the value to be added to each successive line number.

If *increment* is not supplied, the line numbers from the original file are used in the new file.

Example

In this example, EXTENDEDIT copies the specified EDIT file into a new file with a larger extent size. In the new file, the line number of the first line will be 1 and the line number increment will be 1.

```
INT(32) start := 1000D;
INT(32) increment := 1000D;
      .
      .
err := EXTENDEDIT ( filenumber, start, increment );
```

Related Programming Manual

For programming information about the EXTENDEDIT procedure, see the *Guardian Programmer's Guide*.

5 Guardian Procedure Calls (F)

This section contains detailed reference information for all user-accessible Guardian procedure calls beginning with the letter F. Table 5-1 lists all the procedures in this section.

Table 5-1. Procedures Beginning With the Letter F (page 1 of 2)

Table 5-1. Procedures Beginning With the Letter F (page 2 of 2)

FILE_ALTERLIST_ Procedure

Summary

The FILE_ALTERLIST_ procedure changes certain attributes of a disk file that are normally set when the file is created.

Syntax for C Programmers

```
#include <cextdecs(FILE_ALTERLIST_)>

short FILE_ALTERLIST_ ( const char *filename
                       ,short length
                       ,short *item-list
                       ,short number-of-items
                       ,short *values
                       ,short values-length
                       ,[ short partonly ]
                       ,[ short *error-item ] );
```

Syntax for TAL Programmers

```
error := FILE_ALTERLIST_ ( filename:length      ! i:i
                          ,item-list            ! i
                          ,number-of-items      ! i
                          ,values               ! i
                          ,values-length        ! i
                          ,[ partonly ]         ! i
                          ,[ error-item ] );    ! o
```

Parameters

error returned value

INT

is a file-system error number indicating the outcome of the operation.

filename:*length* input:input

STRING .EXT:ref:*, INT:value

specifies the name of the file to be altered. The value of *filename* must be exactly *length* bytes long. It must be a valid disk file name. If the name is partially qualified, it is resolved using the contents of the VOLUME attribute of the =_DEFAULTS DEFINE.

item-list input

INT .EXT:ref:*

is an array that specifies the file attributes for which new values are supplied in the *values* parameter. Each element of the array must be of type INT and must contain a code from Table 5-2 on page 5-5. Some items require the presence of other items and must be supplied in a particular order, as noted in the table.

number-of-items input

INT:value

is the number of items supplied in *item-list*.

values input

INT .EXT:ref:*

is the array in which the values for the file attributes specified in *item-list* are supplied. The values should be supplied in the order specified in *item-list*. Each value begins on an INT boundary; if a value has a length that is an odd number of bytes, then an unused byte should occur before this value begins. The length of each fixed length value is given in Table 5-2 on page 5-5. Every variable length item has an associated item that gives its length, as specified in the table.

values-length input

INT:value

is the size in bytes of *values*.

partonly input

INT:value

for partitioned files, specifies whether the attributes be altered for all partitions (if the supplied value is 0), or just for the named partition (if the value is 1). Nonpartitioned files should use 0. The default is 0.

A value of 1 cannot be specified for some alterations, as noted in Table 5-2. If an alteration would affect alternate-key files, a value of 1 prevents this.

error-item output

INT .EXT:ref:1

if present, returns the index of the item in *item-list* that was being processed when an error was detected, or is one greater than the number of items if an error was detected after the processing of individual items was completed. The index of the first item in *item-list* is 0.

Considerations

- The specified file cannot be open when FILE_ALTERLIST_ is called.
- The caller of FILE_ALTERLIST_ must have read and write access to the specified file.
- If a file attribute already has the value supplied to FILE_ALTERLIST_, no error is returned.
- Except as noted in Table 5-2, the alterations are not made to alternate-key files, but they are made to secondary partitions unless *partonly* is 1.
- If a partition or alternate-key file is not accessible, error 3 or 4 is returned. The accessible partitions or alternate-key files are still altered.

Table 5-2. FILE_ALTERLIST_ Item Codes (page 1 of 8)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects.

Code	Size (bytes)	Description
42	2	File code. For disk objects other than SQL shorthand views, an application-defined value associated with the file. File codes 100 to 999 are reserved for use by HP.
57	8	Expiration time. For disk objects other than SQL shorthand views, the Julian GMT timestamp giving the time before which the file cannot be purged.
65	2	Odd unstructured. For unstructured files, causes the file to allow odd-byte positioning and transfers. The supplied value must be 1. Once this attribute is set for a file, it cannot be reset.
66	2	Audited file. A value of 1 if the file is to be a TMF-audited object; 0 otherwise. Must be 0 for systems without the TMF subsystem. Unless *partonly* is 1, all alternate-key files and all partitions are changed.

Table 5-2. FILE_ALTERLIST_ Item Codes (page 2 of 8)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects.

Code	Size (bytes)	Description
70	2	Refresh EOF. For disk objects other than SQL shorthand views, a value of 1 if a change to the end-of-file value is to cause the file label to be written immediately to disk; 0 otherwise.
78	2	Reset broken flag. Must be 0, indicating that the file is no longer to be marked "broken". For a partitioned file, *partonly* must be 1 when changing this attribute.
80	2	Secondary partition. For disk objects, a value of 0 indicates a primary partition and a value of 1 indicates a secondary partition.

These four items are used for altering the partition description. You can alter the partition description in these ways:

- You can change the volume names of existing partitions.
- For non-key-sequenced files, you can add new partitions.
- For key-sequenced files, you can change the extent sizes of partitions.

These items alter only the partition description in the primary file; no secondary partitions are moved, updated, or created. The *partonly* parameter must be 0 to use these items. You must specify the items in this order: item 90, then item 91 or 97, then item 92 or 98, and finally item 93 or 99.

90	2	Number of partitions. For disk objects, specifies the number of extra (secondary) partitions the file is to have. The maximum value is 15.
91	*	Partition descriptors. An array of 4-byte values, one for each secondary partition: Each entry has this structure: `INT` *primary-extent-size*; `INT` *secondary-extent-size*; These values give the primary and secondary extent sizes in pages (2048-byte units). The length of this item in bytes is four times the value of item 90.
92	*	Partition-volume name-length array. An array of byte counts, each of type INT, giving the length of each partition volume name supplied in item 93.

Table 5-2. FILE_ALTERLIST_ Item Codes (page 3 of 8)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects.

Code	Size (bytes)	Description
93	*	Partition-volume names. A string containing the concatenated names of all the secondary partition volumes, ordered by partition number. You can add new names or change old names; you cannot delete partition volume names. Each name occupies exactly the number of characters specified in the corresponding entry of item 92, hence the total length of this item is the sum of the values in item 92. Partially qualified volume names are resolved using the contents of the caller's =_DEFAULTS DEFINE. The volume name can be a full eight characters (including the dollar sign) only if the system (specified or implied) is the same as the system on which the primary partition resides.

97	*	Partition descriptors (32-bit). An array of 8-byte values, one for each secondary partition. Each entry has this structure:
		`INT (32) priextentsize;`
		`INT (32) secextentsize;`
		These values give the primary and secondary extent sizes in pages. For Format 1 files, the size must be less than 65,535 pages. Format 2 is required for extent sizes over 65,535 pages. The length of this item in bytes is eight times item 90.

98	*	Partition-volume relative names-length array. An array of INT byte counts, each giving the length of the volume-relative name (supplied in item 99) where the corresponding extra partition resides. The length of this item is two times item 90.
99	*	Partition-volume relative names. Concatenated names of the extra partition volumes. Each name occupies the number of characters specified in the corresponding entry of item 98; thus, the total length of this parameter is the sum of the values in item 98. This is an alternate form for item 93 and, if used, must immediately follow item 98. The names can be partially qualified (missing a system name), but the semantics of the names are different from that of item 93. If the system name is missing, the system of the primary file will be used. An implicit system is not recorded explicitly with the file, and so, remains relative to the primary file if copied to another system. The volume name can be eight characters (including "$") only if the specified or implied system is the same as the system where the primary partition is created.

Table 5-2. FILE_ALTERLIST_ Item Codes (page 4 of 8)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects.

Code	Size (bytes)	Description
These eight items alter only the alternate-key description of the primary file; no alternate-key files are purged or created.The *partonly* parameter must be 0 to use these items.You must specify exactly five items if you specify any, and you must specify them consecutively in this order: item 100, item 101 or 106, item 102, item 103 or 108, and finally item 104 or 109.		
100	2	Number of alternate keys. For unstructured files, must be 0.
101	*	Alternate-key descriptors. An array of key-descriptor entries, one for each alternate key. Each entry is 12 bytes long and contains these elements in the order presented here:
	key-specifier (INT:1)	uniquely identifies the alternate-key field. This value is passed to the KEYPOSITION procedure for references to this key field. Must be nonzero.
	key-len (INT:1)	specifies the length, in bytes, of the alternate-key field. The maximum key length of an alternate key that allows duplicates and is defined as insertion-ordered (see *attributes*, below) is: 255 - (10 + primary key length) For unique keys, the maximum is 253. For normal duplicates, the maximum is (253 - primary-key length). For further information, see the *Enscribe Programmer's Guide*.
	key-offset (INT:1)	is the number of bytes from the beginning of the record to where the alternate-key field starts.
	key-filenum (INT:1)	is the relative number in the alternate-key parameter array of this key's alternate-key file. The first alternate-key file's *key-filenum* is 0.
	null-value (INT:1)	Specifies a null value if *attributes*.<0> = 1. Note that the character must reside in the righthand byte.

Table 5-2. FILE_ALTERLIST_ Item Codes (page 5 of 8)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects.

Code	Size (bytes)	Description		
				During a write operation, if a null value is specified for an alternate-key field, and the null value is encountered in all bytes of this key field, the file system does not enter the reference to the record in the alternate-key file. (If the file is read using this alternate-key field, records containing a null value in this field will not be found.)
				During a writeupdate operation (*write-count* = 0), if a null value is specified, and the null value is encountered in all bytes of this key field within *buffer*, the file system deletes the record from the primary file but does not delete the reference to the record in the alternate file.
		attributes (INT:1)		Contains these fields:
		<0>	= 1	means a null value is specified.
		<1>	= 1	means the key is unique. If an attempt is made to insert a record that duplicates an existing value in this field, the insertion is rejected with an error 10 (duplicate record).
		<2>	= 1	means that automatic updating is not performed on this key.
		<3>	= 0	means that alternate-key records with duplicate key values are ordered by the value of the primary-key field. This attribute has meaning only for alternate keys that allow duplicates.
			= 1	means that alternate-key records with duplicate key values are ordered by the sequence in which those records were inserted into the alternate-key file. This attribute has meaning only for alternate keys that allow duplicates.
		<4:15>		Reserved (must be 0)

Table 5-2. FILE_ALTERLIST_ Item Codes (page 6 of 8)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects.

Code	Size (bytes)	Description
		The length in bytes of this item is 12 times the value of item 100.
102	2	Number of alternate-key files. Specifies the number of files that are to hold alternate-key records. The maximum value is 100; the default is 0. FILE_ALTERLIST_ does not automatically create the alternate-key files.
103	*	Alternate-file name-length array. An array of INT values, each giving the length in bytes of the corresponding alternate-file name found in item 104. The length in bytes of this item is 2 times the value of item 102.
104	*	Alternate-file names. A string array containing the concatenated names of the alternate-key files. Since each name occupies exactly the number of characters specified in the corresponding entry of item 103, the total length of this item is the sum of the values in item 103. The names can be fully or partially qualified. Partially qualified names are resolved using the contents of the =_DEFAULTS DEFINE. The volume portion of an alternate-file name can be a full eight characters, including the dollar sign, only if the system (specified or implied) is the same as the system on which the primary file is being created.

Table 5-2. FILE_ALTERLIST_ Item Codes (page 7 of 8)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects.

Code	Size (bytes)	Description

106	*	Alternate-key descriptors (32-bit). An array of 14-byte key descriptor entries, one for each alternate key. Each entry contains this structure:
		`INT key-specifier;`
		`INT key-len;`
		`INT (32) key-offset;`
		`INT key-filenum;`
		`INT null-value;`
		`INT attributes;`
		The *attributes* parameter has these fields:
		<0> Do not index when null.
		<1> Unique.
		<2> Do not update.
		<3> Insertion order duplicates.
		<4:15> Reserved. Must be zero.
		These fields have the same semantics as the corresponding fields of item 101.
		The length of this item in bytes is 14 times item 100. This is an alternate form for item 101, and if used, must immediately follow item 100 in place of item 101.

Table 5-2. FILE_ALTERLIST_ Item Codes (page 8 of 8)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects.

Code	Size (bytes)	Description
108	*	Alternate-file relative name-length array. An array of INT byte counts, each giving the length of the corresponding alternate-file name in item 109. The length of this item is two times item 102. This is an alternate form for item 103, and if used, must immediately follow item 102 in place of item 103.
109	*	Alternate-file relative names. Concatenated names of the alternate-key files. Each name occupies the number of characters specified in the corresponding entry of item 108; total length of this parameter is the sum of the values in item 108. The names must be fully qualified, except the system name can be missing. If the system name is missing, the system of the primary file will be used. Also, an implicit system is not recorded explicitly with the file, and so it remains relative to the primary file if copied to another system. The volume portion of the name can be eight characters (including the "$") only if the specified or implied system is the same as the system where the primary partition is created. This is an alternate form for item 104 and, if used, must immediately follow item 108.
140	8	Partition modification time. For disk objects other than SQL shorthand views, the Julian GMT timestamp indicating the last modification time of the partition named in the open operation (when returned by FILE_GETINFOLIST_) or of the partition named in this call (when returned by FILE_GETINFOLISTBYNAME_).

OSS Considerations

This procedure operates on Guardian objects only. If an OSS file is specified, error 1163 is returned.

Example

```
itemlist := 42;     ! change file code
numberitems := 1;
val := 55;          ! new file code is 55
val^length := 2;
error := FILE_ALTERLIST_ ( fname:length, itemlist,
                           numberitems,
                           val, val^length );
```

Related Programming Manual

For programming information about the FILE_ALTERLIST_ procedure, see the *Guardian Programmer's Guide*.

FILE_CLOSE_ Procedure

Summary

The FILE_CLOSE_ procedure closes an open file. Closing a file terminates access to the file. You can use FILE_CLOSE_ to close files that were opened by either FILE_OPEN_ or OPEN.

Syntax for C Programmers

```
#include <cextdecs(FILE_CLOSE_)>

short FILE_CLOSE_ ( short filenum
                  ,[ short tape-disposition ] );
```

Syntax for TAL Programmers

```
error := FILE_CLOSE_ ( filenum                    ! i
                     ,[ tape-disposition ] );     ! i
```

Parameters

error returned value

INT

is a file-system error number indicating the outcome of the operation. No error is retryable; most of the possible error conditions are the result of programming errors.

filenum input

INT:value

is the number identifying the open file to be closed. *filenum* was returned by FILE_OPEN_ or OPEN when the file was originally opened.

tape-disposition input

INT:value

is one of these values, indicating the tape control action to take:

0 Rewind and unload; do not wait for completion.
1 Rewind and unload, do not wait for completion.
2 Rewind and leave online; do not wait for completion.
3 Rewind and leave online; wait for completion.
4 Do not rewind; leave online.
5 Reserved for parallel backup.

Other input values result in no error if the file is a tape device; the control action might be unpredictable.

If omitted, 0 is used.

Considerations

- Returning space allocation after closing a file

  Closing a disk file causes the space that is used by the resident file control block to be returned to the system main-memory pool if the disk file is not open concurrently.

  A temporary disk file is purged if the file was not open concurrently. Any space that is allocated to that file is made available for other files.

  With any file closure, the space allocated to the access control block (ACB) is returned to the system.

- Closing a nowait file

  If FILE_CLOSE_ is executed for a nowait file that has pending operations, any incomplete operations are canceled. There is no indication as to whether the operation completed or not.

- Labeled tape processing

  If your system has labeled tape processing enabled, all tape actions (as specified by *tape-disposition*) wait for completion.

Messages

- Process close message

  A process can receive a process close system message when it is closed by another process. It can obtain the process handle of the closer by a subsequent

call to FILE_GETRECEIVEINFO_. For detailed information about system messages, see the *Guardian Procedure Errors and Messages Manual*.

Note. This message is also received if the close is made by the backup process of a process pair. Therefore, a process can expect two of these messages when being closed by a process pair.

Related Programming Manuals

For programming information about the FILE_CLOSE_ procedure, see the *Guardian Programmer's Guide*.

FILE_CLOSE_CHKPT_ Procedure

Summary

The FILE_CLOSE_CHKPT_ procedure is called by a primary process to close a designated file in its backup process.

The backup process must be in the *monitor* state (that is, in a call to CHECKMONITOR) for FILE_CLOSE_CHKPT_ to be called successfully. The call to FILE_CLOSE_CHKPT_ causes the CHECKMONITOR procedure in the backup process to call the FILE_CLOSE_ procedure for the designated file.

Syntax for C Programmers

This passive backup procedure is not supported in C programs. For a comparison of active backup and passive backup, see the *Guardian Programmer's Guide*.

Syntax for TAL Programmers

```
error := FILE_CLOSE_CHKPT_ ( filenum                    ! i
                           ,[ tape-disposition ] );     ! i
```

Parameters

error returned value

INT

is a file-system error number indicating the outcome of the checkpoint operation.

filenum input

INT:value

is the number identifying the open file to be closed in the backup process. This value was returned by FILE_OPEN_ or OPEN when the file was originally opened.

tape-disposition input

INT:value

is one of these values, indicating the tape control action to take:

0 Rewind and unload; do not wait for completion
1 Rewind and take offline; do not wait for completion
2 Rewind and leave online; do not wait for completion
3 Rewind and leave online; wait for completion
4 Do not rewind; leave online

Other input values result in an error if the file is a tape device; otherwise they are ignored.

If omitted, 0 is used.

Considerations

- Identification of the backup process

  The system identifies the backup process to be affected by FILE_CLOSE_CHKPT_ from the process's mom field in the process control block (PCB). For named process pairs, this field is automatically set up during the creation of the backup process.

- FILE_CLOSE_CHCKPT_ cannot be called for an SQL/MX object.
- See Considerations on page 5-14.

FILE_COMPLETE[L]_ Procedure

Summary

The FILE_COMPLETE[L]_ procedure completes one previously initiated I/O operation for a Guardian file or returns ready information for one Open System Services (OSS) file. The Guardian or OSS file is from a set of files that was previously enabled for completion by one or more calls to the FILE_COMPLETE_SET_ procedure. Use the FILE_COMPLETEL_ procedure to complete the I/O operation initiated by the SERVERCLASS_SENDL_ procedure. Use the FILE_COMPLETE[L]_ procedure to:

- Wait for the operation to complete on a particular file, on a particular set of files, or on any Guardian file. Execution of the calling process suspends until the completion, or a timeout, occurs. A timeout is not considered a completion.
- Check for the operation to complete on a particular file, on a particular set of files, or on any Guardian file. The call to FILE_COMPLETE[L]_ immediately returns to the calling process, regardless of whether there is a completion. If there is no completion, an error indication is returned.

Only one file is completed with each call. If I/O on a Guardian file is completed or if an OSS file is ready, a structure containing completion information is returned to the caller.

A related procedure, FILE_COMPLETE_GETINFO_, provides information about the set of files that are enabled for completion.

Syntax for C Programmers

```
#include <cextdecs(FILE_COMPLETE_)>

short FILE_COMPLETE_ ( short *completion-info
                     ,[ __int32_t timelimit ]
                     ,[ short *complete-element-list
                     ,[ short num-complete-elements ]
                     ,[ short *error-complete-element ] );


#include <cextdecs(FILE_COMPLETEL_)>

short FILE_COMPLETEL_ ( short _far *completion-info
                      ,[ __int32_t timelimit ]
                      ,[ short _far *complete-element-list
                      ,[ short num-complete-elements ]
                      ,[ short _far *error-complete-element
                        ] );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
status := FILE_COMPLETE_ ( completion-info                  ! o
                         ,[ timelimit ]                     ! i
                         ,[ complete-element-list ]         ! i
                         ,[ num-complete-elements ]         ! i
                         ,[ error-complete-element ] );     ! o


status := FILE_COMPLETEL_ ( completion-info                 ! o
                          ,[ timelimit ]                    ! i
                          ,[ complete-element-list ]        ! i
                          ,[ num-complete-elements ]        ! i
                          ,[ error-complete-element ] );    ! o
```

Parameters

status returned value

INT

is a file-system error number indicating the outcome of the call to the FILE_COMPLETE_ procedure. Error 26 is returned if a *timelimit* value of -1D (indefinite wait) is specified but no I/O operation has been initiated. Error 40 is returned if a *timelimit* value other than -1D is specified and an I/O operation times out; no operation is considered complete (error 40 does not cause an outstanding I/O operation to be canceled).

For cases where there is an error indication on a particular file, see the description of the Z^ERROR field in the structure returned by the *completion-info* parameter. For a description of the fields of this structure, see Structure Definition for *completion-info*.

0 (FEOK)

indicates a successful operation.

completion-info output

INT .EXT:ref:*(ZSYS^DDL^COMPLETION^INFO^DEF)
(Use with FILE_COMPLETE_)

INT .EXT:ref:*(ZSYS^DDL^COMPLETION^INFO2^DEF)
(Use with FILE_COMPLETEL_)

is a structure containing completion information for the Guardian file that was completed or the OSS file that is ready. If an error is returned in the *status* parameter, no information is returned in the *completion-info* parameter. For a description of the fields of the structure, see Structure Definition for *completion-info*.

timelimit input

INT(32):value

specifies whether the FILE_COMPLETE[L]_ procedure is to wait for completion or check for completion. These values of *timelimit* indicate:

>0D	Wait for completion. *timelimit* specifies the maximum time (in 0.01-second units) from the time of the FILE_COMPLETE[L]_ call that the caller can wait for completion.
= 0D	Check for completion. FILE_COMPLETE[L]_ immediately returns to the caller, regardless of whether completion has occurred.
= -1D	Wait indefinitely.
< -1D	An invalid value (file-system error 590 occurs).
omitted	Wait indefinitely.

complete-element-list input

INT .EXT:ref:*(ZSYS^DDL^COMPLETE^ELEMENT^DEF)

is an array of COMPLETE^ELEMENT structures, each structure describing one Guardian file or OSS file. The array explicitly specifies the set of files from which the caller wants the FILE_COMPLETE[L]_ procedure to complete one file. For the current call to FILE_COMPLETE[L]_, this set temporarily overrides the set of files that were enabled for completion by previous calls to the FILE_COMPLETE_SET_ procedure.

Note that the FILE_COMPLETE[L]_ procedure does not perform as efficiently when this temporary override set is used. For better performance, you should use the “permanent” set of enabled files that is established by calling the FILE_COMPLETE_SET_ procedure.

For information on how to set the field values of the COMPLETE^ELEMENT structure, see Structure Definition for COMPLETE^ELEMENT on page 5-28.

num-complete-elements input

INT

is the total number of Guardian files and OSS files specified in the *complete-element-list* parameter. If the *complete-element-list* parameter is supplied, this parameter is required.

error-complete-element output

INT .EXT

returns the index to a COMPLETE^ELEMENT structure in the *complete-element-list* parameter (the temporary override list) that is in error. No file from the temporary override list is completed when there is an error.

An *error-complete-element* value of -1 is returned if no error occurs or if the error does not apply to a particular file in *complete-element-list*.

Structure Definition for *completion-info*

The *completion-info* parameter is a structure that contains completion information for the Guardian file that was completed or the OSS file that is ready.

The structure for the *completion-info* parameter is defined in the ZSYS* files. In the TAL ZSYSTAL file, it is defined as:

```
STRUCT ZSYS^DDL^COMPLETION^INFO^DEF (*);
   BEGIN
   INT      Z^FILETYPE;
   INT(32)  Z^ERROR;
   INT(32)  Z^FNUM^FD;
   STRUCT   Z^RETURN^VALUE;
      BEGIN
      BIT_FILLER   15;
      BIT_FILLER   1;
      BIT_FILLER   13;
      UNSIGNED(1)  Z^READ^READY;
      UNSIGNED(1)  Z^WRITE^READY;
      UNSIGNED(1)  Z^EXCEPTION;
      END;
   INT(32)  Z^COMPLETION^TYPE = Z^RETURN^VALUE;
   INT(32)  Z^COUNT^TRANSFERRED = Z^RETURN^VALUE;
   INT(32)  Z^TAG;
   END;
```

For the FILE_COMPLETEL_ procedure, the structure for the *completion-info2* parameter is defined as:

```
STRUCT ZSYS^DDL^COMPLETION^INFO2^DEF (*);
   BEGIN
   INT      Z^FILETYPE;
   INT(32)  Z^ERROR;
   INT(32)  Z^FNUM^FD;
   STRUCT   Z^RETURN^VALUE;
      BEGIN
      BIT_FILLER   16;
      BIT_FILLER   13;
      UNSIGNED(1)  Z^READ^READY;
      UNSIGNED(1)  Z^WRITE^READY;
      UNSIGNED(1)  Z^EXCEPTION;
      END;
   INT(32)  Z^COMPLETION^TYPE = Z^RETURN^VALUE;
   INT(32)  Z^COUNT^TRANSFERRED = Z^RETURN^VALUE;
   INT(64)  Z^TAG;
   END;
```

In the C ZSYSC file, the structure for the *completion_info* parameter is defined as:

```
typedef struct __zsys_ddl_completion_info {
   short  z_filetype;
   long   z_error;
   long   z_fnum_fd;
   union {
      struct {
         short           filler_0:15;
         unsigned short  filler_1:1;
         short           filler_2:13;
         unsigned short  z_read_ready:1;
         unsigned short  z_write_ready:1;
         unsigned short  z_exception:1;
      } z_return_value;
      unsigned long  z_completion_type;
      long           z_count_transferred;
   }  u_z_return_value;
   long  z_tag;
} zsys_ddl_completion_info_def;
```

Z^FILETYPE

returns the file type of the file that was completed by this call. This field can have these values:

0 The file is a Guardian file.
1 The file is an OSS file.

Z^ERROR

returns a file-system error number for the completion on this file.

Z^FNUM^FD

returns the Guardian file number or OSS file descriptor of the file completed by this call.

Z^COMPLETION^TYPE

returns the type of operation completed on an OSS file. More than one state can be ready. (For Guardian files, this space is replaced by Z^COUNT^TRANSFERRED.) Z^COMPLETION^TYPE contains these fields:

Z^READ^READY

can have these values:

0 Read is not ready for this file.
1 Read is ready for this file.

Z^WRITE^READY

can have these values:

0 Write is not ready for this file.
1 Write is ready for this file.

Z^EXCEPTION

can have these values:

0 No exception occurred for this file.
1 An exception occurred for this file.

Z^COUNT^TRANSFERRED

returns the number of bytes transferred in the completed I/O operation on a Guardian file. (For OSS files, this field is replaced by Z^COMPLETION^TYPE.)

Z^TAG

for Guardian files, returns the application-defined tag that was specified when the completed I/O was initiated. This value is undefined if no tag was supplied for that I/O operation. For OSS files, this field contains 0. For the FILE_COMPLETEL_ procedure, this field is 64 bits.

General Considerations

- Completion on a file by a call to the FILE_COMPLETE[L]_ procedure does not remove it from the set of enabled files. Each file that is enabled for completion is enabled for multiple completions until your program removes it from the enabled set or closes it.
- Files specified in the *complete-element-list* parameter (the temporary override list) must meet the same requirements to be enabled for completion as files specified to the FILE_COMPLETE_SET_ procedure.

Note. For better performance, use the set of files enabled by the FILE_COMPLETE_SET_ procedure rather than specifying a temporary override list to the FILE_COMPLETE[L]_ procedure.

Considerations for Guardian Files

- An application can use the FILE_COMPLETE[L]_ procedure in parallel with the AWAITIOX procedure.
- The “Considerations” for the AWAITIOX procedure generally apply to the FILE_COMPLETE[L]_ procedure. However, note these differences between the FILE_COMPLETE[L]_ and AWAITIOX procedures:
  - The FILE_COMPLETE[L]_ procedure allows you to specify either a particular set of files or all Guardian files for completion (one to be completed by each call).

- The FILE_COMPLETE[L]_ procedure does not provide a way for you to obtain the buffer address associated with an I/O operation or the segment ID of the extended data segment containing the buffer.
- General error conditions are indicated in the return value of the FILE_COMPLETE[L]_ procedure; an error on a particular file is returned in the Z^ERROR field of the *completion^info* structure for that file.
- Error 26 is only returned by the FILE_COMPLETE[L]_ procedure if the caller specified a *timelimit* value of -1D but no I/O operation has been initiated.
- Error 40, which is returned by the FILE_COMPLETE[L]_ procedure if a *timelimit* value other than -1D is specified and an I/O operation times out, does not cause any outstanding I/O operation to be canceled; the operation is considered incomplete.

Considerations for OSS Files

- An application can use the FILE_COMPLETE[L]_ procedure in parallel with the OSS `select()` function.
- Completion on an OSS file means checking for readiness. The file is ready if data can be sent, if data can be received, or if an exception occurred. However, an indication of readiness does not guarantee that a subsequent I/O operation to the file will finish successfully, For example, the ready state of an OSS socket might be changed by another process that shares the socket before your process can initiate its I/O operation.

  The operation of checking for readiness is equivalent to calling the OSS `select()` function, except that the FILE_COMPLETE[L]_ procedure returns ready information for only one file at a time. For additional information, see the `select(2)` function reference page either online or in the *Open System Services System Calls Reference Manual*.
- An OSS child process does not inherit possession of a set of enabled files from the parent process. Membership in a set of enabled files is not propagated to an OSS file that is created by the OSS `dup()` function.
- The FILE_COMPLETE_ operation can be interrupted by an OSS signal. If the calling process receives an OSS signal during the FILE_COMPLETE_ operation, standard signal handling is performed and error 4004 (`EINTR`) might be returned.

Related Programming Manuals

For a general discussion of nowait I/O, see the *Guardian Programmer's Guide*.

Note. The FILE_COMPLETEL_ procedure is supported on systems running J06.07 and later J-series RVUs and H06.18 and later H-series RVUs.

FILE_COMPLETE_GETINFO_ Procedure

Summary

The FILE_COMPLETE_GETINFO_ procedure provides information about the set of files that are currently enabled for completion and thus can be completed by the FILE_COMPLETE_ procedure. These files were enabled for completion by one or more previous calls to the FILE_COMPLETE_SET_ procedure. The information returned includes a list that can contain both Guardian files and Open System Services (OSS) files.

Syntax for C Programmers

```
#include <cextdecs(FILE_COMPLETE_GETINFO_)>

short FILE_COMPLETE_GETINFO_ ( short *info-list
                              ,short maxnum-info-elements
                              ,[ short *num-info-elements ]
);
```

Syntax for TAL Programmers

```
status := FILE_COMPLETE_GETINFO_ (
                               info-list                ! o
                              ,maxnum-info-elements     ! i
                              ,[ num-info-elements ] );  ! o
```

Parameters

status returned value

INT

is a file-system error number indicating the outcome of the operation.

info-list ouput

INT .EXT:ref:*(ZSYS^DDL^COMPLETE^ELEMENT^DEF)

returns an array of COMPLETE^ELEMENT structures, each structure describing one Guardian file or OSS file. The array represents the set of files that were enabled for completion by previous calls to the FILE_COMPLETE_SET_ procedure. This array is compatible for passing directly to the FILE_COMPLETE_

or FILE_COMPLETE_ SET_ procedure. For the definitions of the fields of the COMPLETE^ELEMENT structure, see Structure Definition for COMPLETE^ELEMENT on page 5-28.

maxnum-info-elements input

INT

specifies the maximum number of COMPLETE^ELEMENT structures that can be returned in the *info-list* parameter.

num-info-elements output

INT .EXT

returns the number of COMPLETE^ELEMENT structures that are returned in the *info-list* parameter. If this value is equal to the value specified for *maxnum-info-elements*, there might be additional files that are currently enabled for completion for which no information is being returned.

FILE_COMPLETE_SET_ Procedure

Summary

The FILE_COMPLETE_SET_ procedure enables a set of Guardian and Open System Services (OSS) files for completion by subsequent calls to the FILE_COMPLETE_ procedure. (The FILE_COMPLETE_ procedure completes I/O operations for Guardian files and returns ready information for OSS files.)

The FILE_COMPLETE_SET_ procedure accepts a set of Guardian files and OSS files. You can designate that files be added or removed from the set of files that are enabled

for completion. For OSS files, you can specify the type of operation to be completed (read, write, or exceptions) or change the current specification.

A related procedure, FILE_COMPLETE_GETINFO_, provides information about the set of files that are currently enabled for completion.

Syntax for C Programmers

```
#include <cextdecs(FILE_COMPLETE_SET_)>

short FILE_COMPLETE_SET_ (
                          short *complete-element-list
                         ,short num-complete-elements
                         ,[ short *error-complete-element ] );
```

Syntax for TAL Programmers

```
status := FILE_COMPLETE_SET_ (
                             complete-element-list           ! i
                            ,num-complete-elements           ! i
                            ,[ error-complete-element ] );   ! o
```

Parameters

status returned value

INT

is a file-system error number indicating the outcome of the operation.

complete-element-list input

INT .EXT:ref:*(ZSYS^DDL^COMPLETE^ELEMENT^DEF)

is an array of COMPLETE^ELEMENT structures. Each structure describes one Guardian file or OSS file which the caller wants to add to or remove from the set of files enabled for completion. For information on the fields of the structure, see Structure Definition for COMPLETE^ELEMENT on page 5-28.

num-complete-elements input

INT

is the total number of Guardian files and OSS files specified in the *complete-element-list* parameter.

error-complete-element output

INT .EXT

returns the index to a COMPLETE^ELEMENT structure in the *complete-element-list* parameter indicating the file that is in error. If an error occurs, the call has no affect on the previously established set of files enabled for completion. An *error-complete-element* value of -1 is returned if no error occurs or if the error does not apply to a particular file in *complete-element-list*.

Structure Definition for COMPLETE^ELEMENT

The *complete-element-list* parameter to the FILE_COMPLETE_SET_ and FILE_COMPLETE_ procedures and the *info-list* parameter to the FILE_COMPLETE_GETINFO_ procedure contain arrays of COMPLETE^ELEMENT structures. Each structure describes a Guardian file or OSS file that the caller wants to add to or remove from the set of files enabled for completion.

The COMPLETE^ELEMENT structure is defined in the ZSYS* files. In the TAL ZSYSTAL file, it is defined as:

```
STRUCT ZSYS^DDL^COMPLETE^ELEMENT^DEF (*);
   BEGIN
   INT(32)  Z^FNUM^FD;
   STRUCT   Z^OPTIONS;
      BEGIN
      UNSIGNED(1)  Z^SET^FILE;
      UNSIGNED(1)  Z^FILETYPE;
      BIT_FILLER   14;
      BIT_FILLER   13;
      UNSIGNED(1)  Z^READ^READY;
      UNSIGNED(1)  Z^WRITE^READY;
      UNSIGNED(1)  Z^EXCEPTION;
      END;
   INT(32)  Z^COMPLETION^TYPE = Z^OPTIONS;
   END;
```

In the C ZSYSC file, the `complete_element` structure is defined as:

```
typedef struct __zsys_ddl_complete_element {
   long  z_fnum_fd;
   union {
      struct {
         unsigned short  z_set_file:1;
         unsigned short  z_filetype:1;
         short           filler_0:14;
         short           filler_1:13;
         unsigned short  z_read_ready:1;
         unsigned short  z_write_ready:1;
         unsigned short  z_exception:1;
      } z_options;
      unsigned long  z_completion_type;
   } u_z_options;
} zsys_ddl_complete_element_def;
```

Z^FNUM^FD

is a Guardian file number or OSS file descriptor. Specifying a Guardian file number of -1D for completion indicates that completion of any Guardian file is requested; other specifically enabled Guardian file numbers are ignored. If you have specified a Guardian file number of -1D for completion, you cannot specify any single Guardian file for removal from the set of enabled files; you can only specify Guardian file number -1D. Specifying Guardian file number -1D for removal means that all Guardian files are removed from the set of enabled files.

For OSS file descriptors, the value -1D can be specified only for removal; specifying file descriptor -1D for removal means that all OSS files are removed from the set of enabled files.

For G06.27 and later G-series RVUs, when an OSS terminal file descriptor is passed to this procedure and the terminal process supports the select operation, the procedure adds the OSS terminal file descriptor to the specified read fdset, write fdset, or exception fdset. If the terminal process version does not support the select operation, or when the terminal process supports the select operation but is not set to use the select operation, you get a file-system error. When the terminal process does not support the select operation the system returns error 4216. When the terminal process supports the select operation, but is not set to use the select operation, the system returns error 4219. See the *Guardian Procedure Errors and Messages Manual* for further information on these error numbers.

Z^OPTIONS

specifies attributes for the OSS file or Guardian file. It contains these fields:

Z^SET^FILE

can have these values:

0	Add this file to the set of files that are enabled for completion.
1	Remove this file from the set of files that are enabled for completion.

Z^FILETYPE

can have these values:

0	This file is a Guardian file.
1	This file is an OSS file.

Z^COMPLETION^TYPE

specifies the type of completion desired for an OSS file. (Z^COMPLETION^TYPE is ignored for Guardian files.) It contains these fields:

Z^READ^READY

can have these values:

0	Do not return read ready for this file.
1	Return read ready for this file.

Z^WRITE^READY

can have these values:

0	Do not return write ready for this file.
1	Return write ready for this file.

Z^EXCEPTION

can have these values:

0	Do not return exception occurred for this file.
1	Return exception occurred for this file.

General Considerations

- Each file that is enabled for completion is enabled for multiple completions until your program removes it from the enabled set or closes it. Completion on a file does not remove it from the set of enabled files.
- If the FILE_COMPLETE_SET_ procedure returns an error indication, the request (adding or removing the file from the enabled set) is not performed. Adding a file that is already in the enabled set does not result in an error; the call finishes successfully. Removing a file that is not in the enabled set does not result in an error; the request is ignored and the call finishes successfully.

Considerations for Guardian Files

- A Guardian file specified to the FILE_COMPLETE_SET_ procedure is rejected if it has not been opened in a nowait manner.

Considerations for OSS Files

- An OSS file specified to the FILE_COMPLETE_SET_ procedure is rejected if it is not one of the supported file types. The supported OSS file types are the same as those supported by the OSS `select()` function. For additional information, see the `select(2)` function reference page either online or in the *Open System Services System Calls Reference Manual*.
- OSS files can be opened blocking or nonblocking.
- An OSS child process does not inherit possession of a set of enabled files from the parent process. Membership in a set of enabled files is not propagated to an OSS file that is created by the OSS `dup()` function.

FILE_CREATE_ Procedure

Summary

The FILE_CREATE_ procedure is used to define a new structured or unstructured disk file. The file can be temporary (and therefore automatically deleted when closed) or

permanent. When a temporary file is created, FILE_CREATE_ returns the file name in a form suitable for passing to the FILE_OPEN_ procedure.

Some file characteristics, such as alternate keys and partition information, cannot be specified through this procedure; you must use the FILE_CREATELIST_ procedure to specify them.

Syntax for C Programmers

```
#include <cextdecs(FILE_CREATE_)>

short FILE_CREATE_ ( char *filename
                    ,short maxlen
                    ,short *filenamelen
                    ,[ short file-code ]
                    ,[ short primary-extent-size ]
                    ,[ short secondary-extent-size ]
                    ,[ short maximum-extents ]
                    ,[ short file-type ]
                    ,[ short options ]
                    ,[ short recordlen ]
                    ,[ short blocklen ]
                    ,[ short keylen ]
                    ,[ short key-offset ] );
```

Syntax for TAL Programmers

```
error := FILE_CREATE_ ( filename:maxlen            ! i,o:i
                       ,filenamelen                ! i,o
                       ,[ file-code ]              ! i
                       ,[ primary-extent-size ]    ! i
                       ,[ secondary-extent-size ]  ! i
                       ,[ maximum-extents ]        ! i
                       ,[ file-type ]              ! i
                       ,[ options ]                ! i
                       ,[ recordlen ]              ! i
                       ,[ blocklen ]               ! i
                       ,[ keylen ]                 ! i
                       ,[ key-offset ] );          ! i
```

Parameters

error returned value

INT

is a file-system error number indicating the outcome of the operation.

filename:*maxlen* input, output:input

STRING .EXT:ref:*, INT:value

if a permanent file is to be created, is the name of the new file; if a temporary file is to be created, it is the name of the disk volume on which the file is to be created. If the name is partially qualified, it is resolved using the contents of the caller's =_DEFAULTS DEFINE. If a temporary file is created, the name assigned to the file is returned in *filename*.

maxlen is the length in bytes of the string variable *filename*.

filenamelen input, output

INT .EXT:ref:1

on input, gives the length in bytes of the value supplied in *filename*. On return, it contains the length of the assigned value in *filename* for a temporary file or it is unchanged for a permanent file.

file-code input

INT:value

is an application-defined value to be assigned to the new file. The definition of codes 100 through 999 is reserved for use by HP. The default value is 0.

primary-extent-size input

INT:value

specifies the size of the primary extent in pages (2048-byte units). The value is considered to be unsigned. The system might round the value up to an even number during file creation. The maximum primary extent size is 65,535 (134,215,680 bytes). If this parameter is omitted or equal to 0, 1 is used.

secondary-extent-size input

INT:value

specifies the size of the secondary extents in pages (2048-bytes units). The value is considered to be unsigned. The system might round the value up to an even number during file creation. The maximum secondary extent size is 65,535 (134,215,680 bytes). (The maximum number of secondary extents that a file can have allocated is *maximum-extents* - 1. See *maximum-extents*, following.) If this parameter is omitted or equal to 0, the size of the primary extent is used.

maximum-extents input

INT:value

specifies the maximum number of extents that can be allocated to the new file. The minimum and default value is 16. See "Considerations" for the upper limit on this value.

file-type input

INT:value

specifies the type of structure given the new file. If this parameter is omitted or equal to 0, an unstructured file is created. Valid values are:

0 unstructured
1 relative
2 entry-sequenced
3 key-sequenced

options input

INT:value

specifies optional file attributes. If omitted, the default value of options is 0. The bits, when set, indicate:

<0:8> Reserved (must be 0).

<9> Queue file. The file will be a queue file.

<10> Refresh EOF. A change to the end-of-file value is to cause the file label to be written immediately to disk.

<11> Index compression. For key-sequenced files, the entries in the index blocks are to be compressed. Must be 0 for other file types.

<12> Data compression. For key-sequenced files, the keys of entries in the data blocks are to be compressed. Must be 0 for other file types.

<13> Audit compression. For audited files, the audit data is to be compressed.

<14> Audited. The file is to be audited under the Transaction Management Facility (TMF) subsystem. Must be 0 for systems without the TMF subsystem.

<15> Odd unstructured. For unstructured files, I/O transfers are to occur with the exact counts specified. If this option is not set, transfers are rounded up to an even byte boundary. Must be 0 for other file types. See “Considerations.”

recordlen input

INT:value

for structured files, specifies the maximum number of bytes in a logical record. If omitted, 80 is used. This parameter is ignored for unstructured files. For queue files, this parameter must include 8 bytes for a timestamp.

The formulas for computing the maximum record length (MRL) based on *blocklen* are:

- For relative and entry-sequenced files:

```
MRL = blocklen - 24
```

- For key-sequenced files:

  ```
  MRL = blocklen - 34
  ```

blocklen input

INT:value

for structured files, specifies the length in bytes of each block of records in the file. For unstructured files, it controls the buffer size used internally but does not limit user transfer sizes (though data transfers are more efficient when they do not exceed *blocklen*).

For structured files, *blocklen* must be at least *recordlen* + 24. For key-sequenced files, *blocklen* must be at least *recordlen* + 34.

The system rounds up the specified value to 512, 1024, 2048, or 4096. If omitted, the default size is 4096. Regardless of the specified record length and data-block size, the maximum number of records that can be stored in a data block is 511.

keylen input

INT:value

for key-sequenced files, specifies the length in bytes of the primary-key field. The maximum length is 255. This parameter is required for key-sequenced files. For queue files, this parameter must include 8 bytes for a timestamp.

key-offset input

INT:value

for key-sequenced files, specifies the offset from the beginning of the record to the beginning of the primary-key field. This parameter is required for key-sequenced files. For queue files, the value of this parameter must be 0.

Considerations

- File pointer action

  The end-of-file pointer is set to 0 after the file is created.

- Disk allocation with FILE_CREATE_

  Execution of the FILE_CREATE_ procedure does not allocate any disk area; it only provides an entry in the volume's directory, indicating that the file exists.

- Altering file security

  The file is created with the caller's process file security, which can be examined and set with the PROCESS_SETINFO_ procedure. Once a file has been created, its file security can be altered by opening the file and issuing the appropriate SETMODE and SETMODENOWAIT procedure calls.

- Odd unstructured files

  An odd unstructured file permits reading and writing of odd byte counts and positioning to odd byte addresses.

  If *options*.<15> is 1 and *file-type* is 0, an odd unstructured file is created. In that case, the values of *record-specifier*, *read-count*, and *write-count* are all interpreted exactly; for example, a *write-count* or *read-count* of 7 transfers exactly 7 bytes.

- Even unstructured files

  If *file-type* is 0 and *options*.<15> is 0, an even unstructured file is created. In that case, the values of *read-count* and *write-count* are each rounded up to an even number; for example, a *write-count* or *read-count* of 7 is rounded up to 8, and 8 bytes are transferred.

  An even unstructured file must be positioned to an even byte address; otherwise, the FILE_GETINFO_ procedure returns error 23 (bad address).

  If you use the File Utility Program (FUP) CREATE or HP Tandem Advanced Command Language (TACL) CREATE command to create a file, it creates an even unstructured file by default.

- Upper limit for *maximum-extents*

  There is no guarantee that a file will be created successfully if you specify a value greater than 500 for *maximum-extents*. In addition, FILE_CREATE_ returns error 21 if the values for *primary-extent-size*, *secondary-extent-size*, and *maximum-extents* yield a file size greater than (2**32) - 4096 bytes (approximately four gigabytes) or a partition size greater than 2**31 bytes (two gigabytes).

  In addition, it is not always possible to allocate all of the extents specified by *maximum-extents*. The actual number of extents that can be allocated depends on the amount of space in the file label. If there are alternate keys or partitions, the maximum number of extents allowed is less than 978. If you specify MAXEXTENTS, you must also consider the primary and secondary extent sizes to avoid exceeding the maximum file size.

  For unstructured files on a disk drive in a disk drive enclosure, both *primary-extent-size* and *secondary-extent-size* must be divisible by 14. If you specify file extents that are not divisible by 14 in a FILE_CREATE_ call, the extents are automatically rounded up to the next multiple of 14, and the specified MAXEXTENTS is lowered to compensate. FILE_CREATE_ does not return an error code to indicate this change. You will be aware of the change only if you call FILE_GETINFOLIST_ to verify the extent size and the MAXEXTENTS attributes.

- Disk accesses and the refresh EOF option

  If a disk file has the refresh EOF option set (*options*.<10> = 1), the file label is immediately written to disk each time the end-of-file (EOF) pointer is changed.

(For a description of the refresh EOF option, see the information on unstructured disk files in the *Enscribe Programmer's Guide*.) Depending on the particular application, there can be a significant decrease in processing throughput due to the increased number of disk writes when the refresh EOF option is set.

- Creating a HP NonStop Storage Management Foundation (SMF) file

  When creating a file on a SMF virtual volume, the system chooses the physical volume on which the file will reside. If you want to specify the physical volume, you must create the file using the FILE_CREATELIST_ procedure.

Safeguard Considerations

For information on files protected by Safeguard, see the *Safeguard Reference Manual*.

OSS Considerations

- This procedure operates only on Guardian objects. If an OSS file is specified, error 1163 is returned.
- The OSS `open()` function always opens Guardian files with shared exclusion mode.

Example

```
file^type := 0;
options.<15> := 1;     ! create an odd unstructured file
error := FILE_CREATE_ ( name:buflen, namelen, file^code,
                        pri^ext,,, file^type, options );
```

Related Programming Manuals

For programming information about the FILE_CREATE_ procedure, see the *Enscribe Programmer's Guide* and the *Guardian Programmer's Guide*.

FILE_CREATELIST_ Procedure

Summary

The FILE_CREATELIST_ procedure is used to define a new structured or unstructured disk file. The file can be temporary (and therefore automatically deleted when closed) or permanent. When a temporary file is created, FILE_CREATELIST_ returns the file name in a form suitable for passing to the FILE_OPEN_ procedure.

FILE_CREATELIST_ allows you to specify certain file characteristics, such as alternate keys and partition information, that cannot be specified through the FILE_CREATE_ procedure.

Syntax for C Programmers

```
#include <cextdecs(FILE_CREATELIST_)>

short FILE_CREATELIST_ ( char *filename
                        ,short maxlen
                        ,short *filenamelen
                        ,short *item-list
                        ,short number-of-items
                        ,short *values
                        ,short values-length
                        ,[ short *error-item ] );
```

Syntax for TAL Programmers

```
error := FILE_CREATELIST_ ( filename:maxlen    ! i,o:i
                           ,filenamelen        ! i,o
                           ,item-list          ! i
                           ,number-of-items    ! i
                           ,values             ! i
                           ,values-length      ! i
                           ,[ error-item ] );  ! o
```

Parameters

error returned value

INT

is a file-system error number indicating the outcome of the operation.

filename:*maxlen* input, output:input

STRING .EXT:ref:*, INT:value

if a permanent file is to be created, is the name of the new file; if a temporary file is to be created, it is the name of the disk volume on which the file is to be created. If the name is partially qualified, it is resolved using the contents of the caller's

=_DEFAULTS DEFINE. If a temporary file is created, the name assigned to the file is returned in *filename*.

maxlen is the length in bytes of the string variable *filename*.

filenamelen input, output

INT .EXT:ref:1

on input, gives the length in bytes of the value supplied in *filename*. On return, it contains the length of the assigned value in *filename* for a temporary file or it is unchanged for a permanent file.

item-list input

INT .EXT:ref:*

is an array that specifies the file-creation attributes for which values are supplied in the *values* parameter. Each element of the array must be of type INT and contain an item code from Table 5-3 on page 5-40. Some items require the presence of other items and must be supplied in a particular order, as noted in the table.

number-of-items input

INT:value

is the number of items supplied in *item-list*.

values input

INT .EXT:ref:*

is the array in which the values for the file attributes specified in *item-list* are supplied. The values should be supplied in the order specified in *item-list*. Each value begins on an INT boundary; if a value has a length that is an odd number of bytes, an unused byte should be appendend before this value begins. The length of each fixed-length value is given in Table 5-3 on page 5-40. Every variable-length item has an associated value that gives its length, as specified in the table.

values-length input

INT:value

is the size in bytes of *values*.

error-item output

INT .EXT:ref:1

if present, returns the index of the item in *item-list* that was being processed when an error was detected, or is one greater than the number of items in item-list if an error was detected after the processing of individual items was completed. The index of the first item in *item-list* is 0.

Item Codes

Table 5-3 shows the item codes that can be specified when calling FILE_CREATELIST_.

Table 5-3. FILE_CREATELIST_ Item Codes (page 1 of 12)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects.

Item Code	Size (Bytes)	Description
41	2	File type. For disk objects other than SQL shorthand views, specifies the type of structure the file is to have. This item must occur in the item list before other items whose meanings depend on the file type. Valid values are: 0 unstructured 1 relative 2 entry-sequenced 3 key-sequenced The default value is 0.
42	2	File code. For disk objects other than SQL shorthand views, an application-defined value associated with the file. File codes 100 through 999 are reserved for use by HP. The default value is 0.
43	2	Logical record length. For structured disk objects, the maximum number of bytes in a logical record. The default value is 80. For details, see the *recordlen* parameter of FILE_CREATE_. Item 196 is an alternate form for this item.
44	2	Block length. For structured disk objects, the size of a block of records; for unstructured disk files, controls the internal buffer size. The supplied value is rounded up to 512, 1024, 2048, or 4096. The default value is 4096. For details, see the *blocklen* parameter of FILE_CREATE_. Item 197 is an alternate form for this item.
45	2	Key offset. For key-sequenced disk files, the byte offset from the beginning of the record to the primary-key field. Either this item or item 198 is required for key-sequenced files.
46	2	Key length. For key-sequenced disk files, the maximum number of bytes in the file's primary-key field. The maximum value is 255. Required for key-sequenced files.
47	2	Lock-key length. For key-sequenced files, specifies the generic lock-key length. The length must be between 1 and the key length of the file, or can be 0 to automatically use the key length of the file. The length must be 0 for non-key-sequenced files.

Table 5-3. FILE_CREATELIST_ Item Codes (page 2 of 12)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects.

Item Code	Size (Bytes)	Description
50	2	Primary extent size. For disk objects other than SQL shorthand views, an unsigned integer specifying the size of the first extent, in pages (2048-byte units). The maximum extent size is 65,535 pages (134,215,680 bytes). If this item is omitted or is equal to 0, a size of 1 is used. The value might be rounded up to an even number during file creation. Item 199 is an alternate form for this item.
51	2	Secondary extent size. For disk objects other than SQL shorthand views, an unsigned integer specifying the size of extents after the first one, in pages (2048-byte units). The maximum extent size is 65,535 (134,215,680 bytes). (A file can have up to 15 secondary extents allocated.) If this item is omitted or equal to 0, the primary extent size is used. The supplied value might be rounded up to an even number during file creation.
52	2	Maximum extents. For disk objects other than SQL shorthand views, the maximum number of extents allowed for the file. The minimum and default value is 16. See Considerations on page 5-35. (For partitioned files that are not key-sequenced, the only value permitted is 16.)
57	8	Expiration time. For disk objects other than SQL shorthand views, the Julian GMT timestamp giving the time before which the file cannot be purged.
65	2	Odd unstructured. For unstructured files, a value of 1 specifies that I/O transfers are to occur with the exact counts specified; a value of 0 specifies that transfers are to be rounded up to an even byte boundary. Must be equal to 0 for other file types. The default value is 0. See Considerations on page 5-52.
66	2	Audited file. A value of 1 specifies that the file is to be audited by the Transaction Management Facility (TMF) subsystem; 0 otherwise. Must be 0 for systems without the TMF subsystem. The default value is 0.
67	2	Audit compression. For audited disk objects other than SQL shorthand views, a value of 1 specifies that the audit data is to be compressed; 0 specifies otherwise. The default value is 0.

Table 5-3. FILE_CREATELIST_ Item Codes (page 3 of 12)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects.

Item Code	Size (Bytes)	Description
68	2	Data compression. For key-sequenced disk objects, a value of 1 specifies that the primary keys of data records are to be compressed; 0 specifies otherwise. Must be 0 for other file types. Can be 1 only if key offset (item 45) is 0. The default is 0.
69	2	Index compression. For key-sequenced disk objects, a value of 1 specifies that index block entries are to be compressed; 0 otherwise. Must be 0 for other file types. The default value is 0.
70	2	Refresh EOF. For disk objects other than SQL shorthand views, a value of 1 specifies that a change to the end-of-file value is to cause the file label to be written immediately to disk; 0 specifies otherwise. The default value is 0.
71	2	Create options. For disk objects, specifies miscellaneous file attributes in the form that FILE_CREATE_ accepts and is an alternative to using items 65 to 70. To use this item, specify attributes as described under the *options* parameter of FILE_CREATE_. If you specify any of items 65 to 70, and if you also specify item 71, the last item to appear takes precedence. The default value is 0. See "Considerations" for queue files.
72	2	Write through. For disk objects, a value of 1 specifies write-through caching; a value of 0 specifies that writes to the file are to be buffered. If omitted, 1 is used for unaudited files and 0 is used for audited files. CAUTION: If writes to an unaudited file are buffered, one or more changes to the file can be lost if a failure occurs that affects the disk or disk process.
73	2	Verify writes. For disk objects other than SQL shorthand views, a value of 1 indicates that the file label is to specify that writes to the file should be read back and the data verified; 0 indicates otherwise. If omitted, 0 is used.
74	2	Serial writes. For disk objects other than SQL shorthand views, a value of 1 indicates that the file label is to specify that writes are to be made serially to the mirror when a file resides on a mirrored volume; 0 indicates otherwise. When this is not equal to 1, the system can choose to do either serial or parallel writes. The default value is 0.

Table 5-3. FILE_CREATELIST_ Item Codes (page 4 of 12)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects.

Item Code	Size (Bytes)	Description
80	2	Secondary partition. For disk objects, a value of 0 indicates a primary partition and a value of 1 indicates a secondary partition.

When item 90 is supplied, it must be immediately followed by item 91 or item 97, then by item 92 or item 98, then finally by item 93 or item 99.

90	2	Number of partitions. For disk files, the number of extra (secondary) partitions the file is to have. The maximum value is 15. The default value is 0. The FILE_CREATELIST_ call creates the secondary partitions as well as the primary partition.

91	*	Partition descriptors. For disk files, an array of 4-byte partition descriptors, one for each secondary partition. Each entry has this structure:
		`INT` *`primary-extent-size`*`;`
		`INT` *`secondary-extent-size`*`;`
		These values give the primary and secondary extent sizes in pages (2048-byte units). The length in bytes of this item is 4 times the value of item 90.

92	*	Partition-volume name-length array. For disk files, an array of INT values, each giving the length in bytes of the volume name (supplied in item 93) on which the corresponding secondary partition is to reside. The length in bytes of this item is 2 times the value of item 90.
93	*	Partition-volume names. For disk files, contains the concatenated names of the secondary partition volumes. Because each name occupies exactly the number of characters specified in the corresponding entry of item 92, the total length in bytes of this item is the sum of the values in item 92. A name can be partially qualified, in which case the missing system name is taken from the =_DEFAULTS DEFINE. The volume name can be a full eight characters, including the dollar sign, only if the system (specified or implied) is the same as the system on which the primary partition is being created. Item 99 is an alternate form for this item.

When item 90 is supplied and the file to be created is key-sequenced, items 94 and 95 must also be supplied following item 93 or item 99, and they must be supplied as consecutive items in the order presented here:

Table 5-3. FILE_CREATELIST_ Item Codes (page 5 of 12)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects.

Item Code	Size (Bytes)	Description
94	2	Partition partial-key length. For partitioned key-sequenced disk files, the number of bytes of the primary key that are used to determine which partition of the file contains a particular record. The minimum value is 1.
95	*	Partition partial-key values. For partitioned key-sequenced disk files, the concatenated partial-key values. Because the number of entries is given by item 90 and the size of each entry is given by item 94, the size of item 95 is the product of those two values.

97	*	Partition descriptors (32-bit). An array of 8-byte values, one for each secondary partition. Each entry has this structure:
		`INT (32) priextentsize;`
		`INT (32) secextentsize;`
		These values give the primary and secondary extent sizes in pages. For Format 1 files, the size must be less than 65,536. The length of this item in bytes is eight times item 90. Item 91 is an alternate form for this item.

98	*	Partition-volume relative names-length array. An array of INT byte counts, each giving the length of volume-relative name (supplied in item 99) where the corresponding extra partition resides. The length of this item is two times item 90.
99	*	Partition-volume relative names. Concatenated names of the extra partition volumes. Each name occupies the number of characters specified in the corresponding entry of item 98; thus, the total length of this parameter is the sum of the values in item 98. This is an alternate form for item 93 and, if used, must immediately follow item 98. The names can be partially qualified (missing a system name) but the semantics of the names are different from that of item 93. A missing system name causes the use of the system where the primary file is created. An implicit system is not recorded explicitly with the file, it remains relative to the primary file if copied to another system. The volume name can be eight characters (including "$") only if the specified or implied system is the same as the system where the primary partition is created.

Table 5-3. FILE_CREATELIST_ Item Codes (page 6 of 12)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects.

Item Code	Size (Bytes)	Description
When item 100 is supplied, it must be immediately followed by a descriptor item (which can be either item 101 or item 106), then by item 102, then by a pair of file-name items (which can be either items 103 and 104, or items 108 and 109).		
100	2	Number of alternate keys in a disk file. For unstructured files, must be 0. The default value is 0.
101	*	Alternate-key descriptors. For disk files, an array of key-descriptor entries, one for each alternate key. Each entry is 12 bytes long and contains these elements in the order presented here:
		key-specifier (INT:1) — uniquely identifies the alternate-key field. This value is passed to the KEYPOSITION procedure for references to this key field. Must be nonzero.
101 (continued)		*key-len* (INT:1) — specifies the length in bytes of the alternate-key field. The maximum key length of an alternate key that allows duplicates and is defined as insertion-ordered (see *attributes*, later) is: 255 - (10 + primary-key length) For unique keys, the maximum length is 253. For normal duplicates, the maximum length is (253 - primary-key length). For further information about maximum key length, see the *Enscribe Programmer's Guide*.
		key-offset (INT:1) — is the number of bytes from the beginning of the record to where the alternate-key field starts.
		key-filenum (INT:1) — is the relative number in the alternate-key parameter array of this key's alternate-key file. The first alternate-key file's *key-filenum* is 0.

Table 5-3. FILE_CREATELIST_ Item Codes (page 7 of 12)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects.

Item Code	Size (Bytes)	Description		
		null-value (INT:1)	specifies a null value if *attributes*.<0> = 1. Note that the character must reside in the right-hand byte. During a write operation, if a null value is specified for an alternate-key field, and if the null value is encountered in all bytes of this key field, the file system does not enter the reference to the record in the alternate-key file. (If the file is read using this alternate-key field, records containing a null value in this field will not be found.) During a writeupdate operation (*write-count* = 0), if a null value is specified, and if the null value is encountered in all bytes of this key field within *buffer*, the file system deletes the record from the primary file but does not delete the reference to the record in the alternate file.	
		attributes (INT:1)	contains these fields:	
		<0>	= 1	means a null value is specified.
		<1>	= 1	means the key is unique. If an attempt is made to insert a record that duplicates an existing value in this field, the insertion is rejected with an error 10 (duplicate record).
101 (continued)		<2>	= 1	means that automatic updating cannot be performed on this key.
		<3>	= 0	means that alternate-key records with duplicate key values are ordered by the value of the primary-key field. This attribute has meaning only for alternate keys that allow duplicates.

Table 5-3. FILE_CREATELIST_ Item Codes (page 8 of 12)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects.

Item Code	Size (Bytes)	Description
		= means that alternate-key records with 1 duplicate key values are ordered by the sequence in which those records were inserted into the alternate-key file. This attribute has meaning only for alternate keys that allow duplicates. <4:15> Reserved (specify 0) The length in bytes of this item is 12 times the value of item 100.
102	2	Number of alternate-key files. For disk files, specifies the number of files that are to hold alternate-key records. The maximum value is 100; the default value is 0. FILE_CREATELIST_ does not automatically create the alternate-key files.
103	*	Alternate-file name-length array. For disk files, an array of INT values, each giving the length in bytes of the corresponding alternate-file name found in item 104. The length in bytes of this item is 2 times the value of item 102.
104	*	Alternate-file names. For disk files, a string array containing the concatenated names of the alternate-key files. Because each name occupies exactly the number of characters specified in the corresponding entry of item 103, the total length of this item is the sum of the values in item 103. The names can be fully or partially qualified. Partially qualified names are resolved using the contents of the =_DEFAULTS DEFINE. The volume portion of an alternate-file name can be a full eight characters, including the dollar sign, only if the system (specified or implied) is the same as the system on which the primary file is being created. Item 109 is an alternate form for this item.

Table 5-3. FILE_CREATELIST_ Item Codes (page 9 of 12)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects.

Item Code	Size (Bytes)	Description

106	*	Alternate-key descriptors (32-bit). An array of 14-byte key descriptor entries, one for each alternate key. Each entry has this structure:
		`INT key-specifier;`
		`INT key-len;`
		`INT (32) key-offset;`
		`INT key-filenum;`
		`INT null-value;`
		`INT attributes;`
		attributes has these fields:
		<0> Do not index when null.
		<1> Unique.
		<2> Do not update.
		<3> Insertion order duplicates.
		<4:15> Reserved. Must be zero.
		These fields have the same semantics as the corresponding fields of item 101.
		The length of this item in bytes is 14 times item 100. This is an alternate form for item 101, and if used, must immediately follow item 100 in place of item 101.

108	*	Alternate-file relative name-length array. An array of INT byte counts, each giving the length of the corresponding alternate-file name in item 109. The length of this item is two times item 102. This is an alternate form for item 103, and if used, must immediately follow item 102 in place of item 103.

Table 5-3. FILE_CREATELIST_ Item Codes (page 10 of 12)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects.

Item Code	Size (Bytes)	Description
109	*	Alternate-file relative names. Concatenated names of the alternate-key files. Each name occupies the number of characters specified in the corresponding entry of item 108; total length of this parameter is the sum of the values in item 108. The names must be fully qualified, except the system name can be missing. If the system name is missing, the system of the primary file is used. Also, an implicit system is not recorded explicitly with the file, and so it remains relative to the primary file if copied to another system. The volume portion of the name can be eight characters (including the "$") only if the specified or implied system is the same as the system where the primary partition is created. This is an alternate form for item 104 and, if used, must immediately follow item 108.
The items 178-179 must both be supplied if either is supplied, and they must be supplied as consecutive items in the order presented here:		
178	2	Physical volume name length. The length in bytes of the name given by item 179.
179	*	Physical volume name. When creating a file on a SMF virtual volume, this item specifies the physical volume on which the file is to reside. The name is in external form and optionally includes the node name; if the node name is unspecified, it is taken from the =_DEFAULTS DEFINE. The physical volume must be a member of the pool associated with the virtual volume. If this item is not supplied, the SMF subsystem chooses a physical volume from the pool.
180	2	Suggested primary processor. If not -1 (null), and if there is a choice of suitable physical volumes, this value specifies the processor number of the processor desired to contain the primary process of the disk process providing the physical storage for the file. This value is advisory only and does not have any effect if the file is not being created on a SMF virtual volume, if a physical volume is specified by using item 179, or if no suitable physical volume is available on which a virtual disk process in the specified processor could place a file.

Table 5-3. FILE_CREATELIST_ Item Codes (page 11 of 12)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects.

Item Code	Size (Bytes)	Description
The items 187 and188 are optional when creating a partitioned SMF file. These items must both be supplied if either is supplied; they must be supplied at some point after item 90 (number of partitions) and must be supplied as consecutive items in the order presented here:		
187	*	Partition physical volume name-length array. For SMF files, an array of INT values that give, for each extra (secondary) partition, a length for the physical volume name, which is supplied in item 188. If a length is 0, the corresponding extra partition has its physical volume selected by the SMF subsystem. A length must be 0 if the corresponding partition volume (in item 93) is not a SMF virtual volume. The length of this item is 2 times the number of partitions (item 90).
188	*	Partition physical volume names. For SMF files, contains the concatenated names of the secondary partition physical volumes. Each name occupies the number of characters specified in the corresponding entry of item 187; the total length of this item equals the sum of the values in item 187. The names can be partially qualified, in which case the missing node name is taken from the =_DEFAULTS DEFINE. Each physical volume must be a member of the pool associated with the corresponding partition volume given in item 93.
195	2	File format. File format can be 0, 1, or 2. Format 1 files allow only as many as 4 KB blocks and as many as 2 GB partitions. Format 2 files allow larger blocks and partitions. The value 0 (the default) specifies that the system select the format based on the values of other parameters.
196	4	Logical record length (32-bit). For structured disk files, the maximum number of bytes in a logical record. If omitted, 80 is used. This is an alternate form for item 43.
197	4	Block length (32-bit). For structured files, the size of a block of records. For unstructured files, the unstructured buffer size. Currently, the maximum supported value, which is also the default value, is 4096. This is an alternate form for item 44.

Table 5-3. FILE_CREATELIST_ Item Codes (page 12 of 12)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects.

Item Code	Size (Bytes)	Description
198	4	Key offset (32-bit). For key-sequenced disk files, the byte offset from the beginning of the record to the primary key field. This an alternate form for item 45; one of the two items is required for key-sequenced files.
199	4	Primary extent size (32-bit). The size in pages (2048-byte units) of the first extent. If omitted or 0, a size of 1 is used. The value can be rounded during creation to a multiple of the block size. For Format 1 files, the size must be less than 65,536. This is an alternate form for item 50.
200	4	Secondary extent size (32-bit). The size in pages (2048-byte units) of extents after the first extent. If 0 or omitted, the size of the primary extent is used. The value can be rounded during creation to a multiple of the block size. For Format 1 files, the size must be less than 65,536. This is an alternate form for item 51.
212	2	Block checksumming option. For Format 2 structured files, specifies whether data protection needs to be used through checksum calculation and comparison (1 specified if needed and 0 if not needed). If omitted or -1 is specified, the default value 1 is used. This item is ignored for files that do not support a checksum option.
221	*	Partition maximum extents array. This item is an array of INT values that specify the maximum extents for each secondary partition. This item is optional and can only be specified for key-sequenced files. If present, it must be supplied at some point after item 90 (number of partitions) and item 41 (file type). If this item is not present, FILE_CREATELIST_ will use the maximum extents value for the file (item 52) and apply it to each secondary partition. The length of the item is 2 times the number of partitions (item 90).

Considerations

- Using *item-list*

  In general, if you supply an item in *item-list* that is not applicable (for example, if you supply a logical record length for an unstructured file), FILE_CREATELIST_ ignores the item as long as it passes syntax and value range checks. Exceptions to this are noted in Table 5-3 on page 5-40.

- File pointer action

  The end-of-file pointer is set to 0 after the file is created.

- Disk allocation with FILE_CREATELIST_

  Execution of the FILE_CREATELIST_ procedure does not allocate any disk area; it only provides an entry in the volume's directory, indicating that the file exists.

- Altering file security

  The file is created with the caller's process file security which can be examined and set with the PROCESSFILESECURITY procedure. Once a file has been created, its file security can be altered by opening the file and issuing the appropriate SETMODE and SETMODENOWAIT procedure calls.

- Odd unstructured files

  An odd unstructured file permits reading and writing of odd byte counts and positioning to odd byte addresses.

  If item 65 is set to 1 and item 42 is set to 0 in *item-list*, an odd unstructured file is created. In that case, the values of *record-specifier*, *read-count*, and *write-count* are all interpreted exactly; for example, a *write-count* or *read-count* of 7 transfers exactly 7 bytes.

- Even unstructured files

  If items 65 and 42 are both set to 0 in *item-list*, an even unstructured file is created. In that case, the values of *read-count* and *write-count* are each rounded up to an even number; for example, a *write-count* or *read-count* of 7 is rounded up to 8, and 8 bytes are transferred.

  An even unstructured file must be positioned to an even byte address; otherwise, the FILE_GETINFO_ procedure returns error 23 (bad address).

  If you use the File Utlity Program (FUP) CREATE or HP Tandem Advanced Command Language (TACL) CREATE command to create a file, it creates an even unstructured file by default.

- Upper limit for *maximum-extents*

  If you specify a value greater than 500 for *maximum-extents* (item 52 in *item-list*), there is no guarantee that a file will be created successfully. In addition, FILE_CREATELIST_ returns error 21 if the values for primary and secondary

extent sizes (items 50, 51, and 91 in *item-list*) and *maximum-extents* (item 52) yield a file size greater than (2**32) - 4096 bytes (approximately four gigabytes) or a partition size greater than 2**31 bytes (two gigabytes).

For unstructured files on a disk drive in a disk drive enclosure, both *primary-extent-size* and *secondary-extent-size* must be divisible by 14. If you specify file extents that are not divisible by 14 in a FILE_CREATE_ call, the extents are automatically rounded up to the next multiple of 14, and the specified MAXEXTENTS is lowered to compensate. FILE_CREATE_ does not return an error code to indicate this change. You will be aware of the change only if you call FILE_GETINFOLIST_ to verify the extent size and the MAXEXTENTS attributes.

- Insertion-ordered alternate keys

  All the nonunique alternate keys of a file must have the same duplicate-key-ordering attribute. That is, a file cannot have both insertion-ordered alternate keys and standard (duplicate ordering by primary key) nonunique alternate keys. An insertion-ordered alternate key cannot share an alternate key file with other keys of different lengths or with other keys that are not insertion-ordered.

  The FILE_CREATELIST_ procedure returns error 46 if the rules of usage for insertion-ordered alternate keys are violated.

  When an alternate-key record is updated, the timestamp portion of the key is also updated. Alternate-key records are updated only when the corresponding alternate-key field of the primary record is changed.

  The relative position of an alternate-key record within a set of duplicates might change if an unrecoverable error occurs during a writeupdate of the primary record.

  There is a performance penalty for using insertion-ordered duplicate alternate keys. Updates and deletes of alternate-key fields force the disk process to sequentially search the set of alternate-key records having the same alternate key value until a match is found on the primary-key-value portion of the key. (The value of the timestamp field in an alternate key record is not stored in the primary record.) The performance cost rises as the number of records having duplicate alternate-key values increases.

  If an insertion-ordered alternate-key file is partitioned, the length of each partition key should be no greater than the total of the alternate-key tag length and the alternate-key length. If the length of any partition key is greater than this sum, then the file system might fail to advise the user of the duplicate-key condition (indicated by the warning error code 551).

- Queue files

  To create a queue file, specify item 71 as described under the “FILE_CREATE_ Procedure.” For more information, see FILE_CREATE_ Procedure. Some item codes are incompatible with queue files; no partitions or alternate keys can be defined for queue files.

OSS Considerations

This procedure operates only on Guardian objects. If an OSS file is specified, error 1163 is returned.

Related Programming Manuals

For programming information about the FILE_CREATELIST_ procedure, see the *Enscribe Programmer's Guide* and the *Guardian Programmer's Guide*.

FILE_GETINFO_ Procedure

Summary

The FILE_GETINFO_ procedure obtains a limited set of information about a file identified by file number.

A related procedure, FILE_GETINFOLIST_, obtains detailed information about a file identified by file number.

Syntax for C Programmers

```
#include <cextdecs(FILE_GETINFO_)>

short FILE_GETINFO_ ( short filenum
                     ,[ short *last-error ]
                     ,[ char *filename ]
                     ,[ short maxlen ]
                     ,[ short *filename-length ]
                     ,[ short *type-info ]
                     ,[ short *flags ] );
```

- The parameter *maxlen* specifies the maximum length in bytes of the character string pointed to by *filename*, the actual length of which is returned by

filename-length. All three of these parameters must either be supplied or be absent.

Syntax for TAL Programmers

```
error := FILE_GETINFO_ ( filenum                    ! i
                        ,[ last-error ]             ! o
                        ,[ filename:maxlen ]        ! o:i
                        ,[ filename-length ]        ! o
                        ,[ type-info ]              ! o
                        ,[ flags ] );               ! o
```

Parameters

error returned value

INT

is a file-system error number indicating the outcome of the call to FILE_GETINFO_.

filenum input

INT:value

is a number that identifies the open file of interest. *filenum* was returned by FILE_OPEN_ or OPEN when the file was originally opened.

You can also specify -1 for *filenum* to obtain the *last-error* value resulting from a file operation that is not associated with a file number. See “Considerations” below.

last-error output

INT .EXT:ref:1

returns the file-system error number resulting from the last operation performed on the specified file.

filename:*maxlen* output:input

STRING .EXT:ref:*, INT:value

returns the fully-qualified name under which the specified file was opened. If a DEFINE name was supplied when opening the file, *filename* is the file name, not the DEFINE name. *maxlen* gives the length in bytes of the string variable *filename*.

filename-length output

INT .EXT:ref:1

is the length in bytes of the name returned in *filename*.

type-info output

INT .EXT:ref:5

returns an array of INT values that contain information about the file. The meanings of these words are:

[0] Device type

[1] Device subtype

[2:4] The meanings of these words depend on the device type. When the device type is 3 (disk) the meanings are:

[2] Object type. For disk files, a value greater than 0 indicates an SQL object; 0 indicates a nonSQL file. -1 is returned for nondisk files.

[3] File type. For disk files, indicates the file type:

0 unstructured
1 relative
2 entry-sequenced
3 key-sequenced
-1 is returned for nondisk files.

[4] File code. For disk files, gives the application-defined file code (file codes 100-999 are reserved for use by HP). -1 is returned for nondisk files.

flags output

INT .EXT:ref:1

returns additional information about the file. The bits, when set to 1, indicate:

<0:14> Reserved and undefined.

<15> File is an OSS file.

Considerations

- You can obtain the *last-error* value resulting from a file operation that is not associated with a file number by specifying a *filenum* value of -1 to FILE_GETINFO_. An error number can be obtained in this manner for such operations as a purge, waited open, or failed create operation. The result of a preceding awaitio[x] or alter operation can also be obtained in this manner.
- When -1 is supplied for *filenum*, only *last-error* returns useful information. A *filename-length* of 0 is returned.

- If FILE_GETINFO_ is called subsequent to a file close, an *error* value of 16 (file not open) is returned.

OSS Considerations

- Use the *flags* parameter of FILE_GETINFO_ or FILE_GETINFOBYNAME_ or use item code 161 of FILE_GETINFOLIST_ or FILE_GETINFOLISTBYNAME_ to determine whether the file is an OSS file.
- Use the *item-list* parameter of FILE_GETINFOLIST_ or FILE_GETINFOLISTBYNAME_ to specify which OSS file attribute values are to be returned.

Example

```
error := FILE_GETINFO_ ( filenum, lasterror );   ! obtain
                                                 ! error from
                                                 ! last file
                                                 ! operation
```

Related Programming Manuals

For programming information about the FILE_GETINFO_ procedure, see the *Guardian Programmer's Guide*. For information on the SQL objects and programs, see the *HP NonStop SQL/MP Programming Manual for C* and the *HP NonStop SQL/MP Programming Manual for COBOL*.

FILE_GETINFOBYNAME_ Procedure

Summary

The FILE_GETINFOBYNAME_ procedure obtains a limited set of information about a file identified by file name.

A related procedure, FILE_GETINFOLISTBYNAME_, obtains detailed information about a file identified by file name.

Syntax for C Programmers

```
#include <cextdecs(FILE_GETINFOBYNAME_)>

short FILE_GETINFOBYNAME_ ( const char *filename
                           ,short length
                           ,[ short *type-info ]
                           ,[ short *physical-recordlen ]
                           ,[ short options ]
                           ,[ __int32_t tag-or-timeout ] )
                           ,[ short *flags ] );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
error := FILE_GETINFOBYNAME_ ( filename:length          ! i:i
                              ,[ type-info ]            ! o
                              ,[ physical-recordlen ]   ! o
                              ,[ options ]              ! i
                              ,[ tag-or-timeout ]       ! i
                              ,[ flags ] );             ! o
```

Parameters

error returned value

INT

is a file-system error number indicating the outcome of the operation.

filename:*length* input:input

STRING .EXT:ref:*, INT:value

specifies the name of the file of interest. The value of *filename* must be exactly *length* bytes long and must be a valid file name or DEFINE name. If the name is partially qualified, it is resolved using the contents of the =_DEFAULTS DEFINE.

type-info output

INT .EXT:ref:5

returns an array of INT values that contain information about the file. The meanings of these words are:

[0] Device type

[1] Device subtype

[2:4] The meanings of these words depend on the device type. When the device type is 3 (disk) the meanings are:

[2] Object type. For disk files, a value greater than 0 indicates an SQL object; 0 indicates a nonSQL file. -1 is returned for nondisk files.

[3] File type. For disk files, indicates the file type:

0 unstructured
1 relative
2 entry-sequenced
3 key-sequenced
-1 is returned for nondisk files.

[4] File code. For disk files, gives the application-defined file code (file codes 100-999 are reserved for use by HP). -1 is returned for nondisk files.

If an *error* value of 11 (file not found) is returned, the device type and subtype values correctly reflect the device portion of the supplied file name, but the other fields in *type-info* do not contain valid information.

physical-recordlen output

INT .EXT:ref:1

returns the physical record length associated with the file:

nondisk *physical-recordlen* is the configured devices record length.

disk files *physical-recordlen* is the maximum possible transfer length. Transfer length is equal to the configured buffer size for the device (either 2048 or 4096 bytes). (For an Enscribe disk file, the logical record length can be obtained by a call to FILE_GETINFOLIST[BYNAME]_.)

processes and $RECEIVE file
A length of 132 is returned in *physical-recordlen*. This is the system convention for interprocess files.

options input

INT:value

specifies the options desired. The bits, when set to 1, indicate:

<0:12> Reserved (specify 0)

<13> specifies that this call is only initiating a nowait inquiry and the information will be returned in a system message. Do not set both *options*.<13> and *options*.<14>. See “Considerations.”

<14> specifies that the sending of a device type inquiry message to a subtype 30 process should not be allowed to take longer than indicated by *tag-or-timeout*. If the time is exceeded, error 40 is returned.

<15> specifies that device type inquiry messages are not to be sent to subtype 30 processes.

If omitted, 0 is used.

tag-or-timeout input

INT(32):value

is a parameter with two functions depending on the value of *options* that is specified:

- When *options*.<13> is 1, it is a value you supply to help identify one of several FILE_GETINFOBYNAME_ operations. The system stores this value until the operation completes, then returns it to the program in words 1 and 2 of a system message. See “Considerations” below.
- When *options*.<14> is 1, it is the maximum amount of time to wait, expressed in 0.01-second units. The value -1D means wait forever. If the parameter is omitted, -1D is used.

flags output

INT .EXT:ref:1

returns additional information about the file. The bits, when set to 1, indicate:

<0:14> Reserved and undefined.

<15> File is an OSS file.

Considerations

- Specifying a subtype 30 process

  When FILE_GETINFOBYNAME_ is called with a file name that designates a subtype 30 process, the procedure sends a device type inquiry system message to the process to determine the device type and subtype (unless disabled by *options*.<15>). The message sent by FILE_GETINFOBYNAME_ is either in D-series-format (message -106) or C-series-format (message -40) depending on the options used when the subtype 30 process opened $RECEIVE. The formats of these completion messages are described in the *Guardian Procedure Errors and Messages Manual*.

  The subtype 30 process replies with the requested information in system message -106 or -40, corresponding to the message used in the inquiry. The returned device type value should be one of those listed in Appendix A, Device Types and Subtypes. If the message response is incorrectly formatted, the FILE_GETINFOBYNAME_ caller receives device type and subtype values of 0. The REPLY caller (the subtype 30 process) receives an error 2.

  A deadlock occurs if a subtype 30 process calls FILE_GETINFOBYNAME_ on its own process name.

- Using the nowait option

  If you call FILE_INFOBYNAME_ procedure in a nowait manner, the results are returned in the nowait FILE_GETINFOBYNAME_ completion message (-108), not in the output parameters of the procedure. The format of this completion message is described in the *Guardian Procedure Errors and Messages Manual*. If *error* is not 0, no completion message is sent to $RECEIVE. Errors can be reported either on return from the procedure, in which case *error* might be meaningful, or through the completion message sent to $RECEIVE.

  The system reports a path error only after automatically making retries.

  When the nowait option is used, any step of the inquiry operation might be asynchronous to the caller. However, only simulation inquiries to subtype 30 processes are guaranteed to be asynchronous.

  When a process pair uses the nowait option, the nowait FILE_GETINFOBYNAME_ completion message is sent only to the process that made the call, not to the other member of the pair.

  Switching ownership from the primary to the backup process can leave outstanding inquiries. The CHECKSWITCH procedure automatically discards these as it becomes the backup process.

OSS Considerations

- Use the *flags* parameter of FILE_GETINFO_ or FILE_GETINFOBYNAME_ or use item code 161 of FILE_GETINFOLIST_ or FILE_GETINFOLISTBYNAME_ to determine whether the file is an OSS file.
- Use the *item-list* parameter of FILE_GETINFOLIST_ or FILE_GETINFOLISTBYNAME_ to specify which OSS file attribute values are to be returned.

Example

```
error := FILE_GETINFOBYNAME_ ( name:length,typeinfo,reclen );
```

Related Programming Manuals

For programming information about the FILE_GETINFOBYNAME_ procedure, see the *Guardian Programmer's Guide*. For information on the SQL objects and programs, see the *HP NonStop SQL/MP Programming Manual for C* and the *HP NonStop SQL/MP Programming Manual for COBOL*.

FILE_GETINFOLIST_ Procedure

Summary

The FILE_GETINFOLIST_ procedure obtains detailed information about a file identified by file number.

A related (and simpler to use) procedure, FILE_GETINFO_, obtains a limited set of information about a file identified by file number.

Syntax for C Programmers

```
#include <cextdecs(FILE_GETINFOLIST_)>

short FILE_GETINFOLIST_ ( short filenum
                         ,short *item-list
                         ,short number-of-items
                         ,short *result
                         ,short result-max
                         ,[ short *result-length ]
                         ,[ short *error-item ] );
```

Syntax for TAL Programmers

```
error := FILE_GETINFOLIST_ ( filenum              ! i
                            ,item-list            ! i
                            ,number-of-items      ! i
                            ,result               ! o
                            ,result-max           ! i
                            ,[ result-length ]    ! o
                            ,[ error-item ] );    ! o
```

Parameters

error returned value

INT

is a file-system error number indicating the outcome of the operation.

filenum input

INT:value

is a number that identifies the open file of interest. *filenum* was returned by FILE_OPEN_ or OPEN when the file was originally opened.

You can also specify -1 for *filenum* to obtain the *last-error* value resulting from a file operation that is not associated with a file number. See “Considerations.”

item-list input

INT .EXT:ref.*

is an array of values that specify the items of information to be returned by the procedure. Each element of the array must be of type INT and contain a code from Table 5-4 on page 5-66.

number-of-items input

INT:value

specifies the number of items supplied in *item-list*.

result output

INT .EXT:ref:*

is the buffer in which the requested items of information are returned. The item values are returned in the order specified in *item-list*. Each item begins on an INT boundary. Every variable-length item has an associated item giving its length; the caller should put this associated item into *item-list* immediately before the variable length item.

result-max input

INT:value

specifies the maximum size in bytes of the array of values that can be returned in *result*. If the specified size is not large enough to hold the requested items, an *error* value of 563 (buffer too small) is returned.

result-length output

INT .EXT:ref:1

returns the length in bytes of the array of values returned in *result*. *result-length* is an odd value only if the last value in the array has an odd length.

error-item output

INT .EXT:ref:1

returns the index of the item that was being processed when an error was detected. The index of the first item in *item-list* is 0.

Considerations

- Normally if an error is returned, the contents of the *result* parameter are undefined. However, if the returned error code is 2 (operation invalid for file type), the *result* parameter contains a combination of correct values (for valid items) and unchanged memory locations (for invalid items because of the kind of file). The *error-item* value points to the first invalid item.

When error 2 occurs, any items prior to the one pointed to by *error-item* are returned with correct values in the *result* parameter; following items might or might not be valid. If a following item is known to be valid because of the kind of file involved, the correct *result* value for the item can be accessed in the corresponding location in the *result* buffer. To do so, the program will have to account for space in the buffer reserved for preceding invalid items as well as for space for preceding valid items. (Preceding in this case refers to *some* item that occurs before the item in question.) See description of items in Table 5-4 on page 5-66 to determine the kinds of files for which an item is valid.

Invalid items that are fixed-size will have the amount of space reserved in the result buffer, but that section of buffer will be unchanged. Invalid items that are variable-sized have no space reserved for them, but this should not be depended upon because they could become valid in a future RVU and thus start occupying space. The programmer might want to place all the items that could cause error 2 in the item list after those that are not expected to cause this error.

- If a file number for which information is being retrieved was opened with the unstructured access option, the provided information appears as if the file is an unstructured file without partitions or alternate keys.
- The file system stores error information for the last operation that was not associated with a file number (such as a purge, waited open, or failed create operation; the result of a preceding awaitio[x] or alter operation is also stored). You can obtain this stored information from FILE_GETINFOLIST_ by supplying a value of -1 for *filenum*. The error information is returned in items 7 through 10. Valid values are also returned for items 19 and 20.
- The FILE_GETINFOLIST_ procedure should not be used to determine, the *current-key-value* parameter for queue files (item code 15), because a current key position is not maintained for queue files.
- Support for SQL files includes both Format 1 and Format 2 files.
- If the file being referenced is a Format 2 file and the extent size exceeds 65535, item codes will return -1 with no error indication.
- For all items in Table 5-4 on page 5-66 that return some form of last modification time, creation time is returned for an object that has never been modified. Similarly, for items that return some form of last open time, creation time is returned for an object that has never been opened.

Table 5-4 on page 5-66 shows the item codes used by FILE_GETINFOLIST_. Item codes of 30 and greater, except item codes 201 through 206, are also used by FILE_GETINFOLISTBYNAME_.

Table 5-4. FILE_GETINFOLIST_ Item Codes (page 1 of 24)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects. Items described as "Applies only to SQL/MX objects" return file-system error 2 if queried on anything that is not an SQL/MX object.

Item Code	Size (Bytes)	Description
1	2	File-name length. The length in bytes of the file name returned by item 2.
2	*	File name. The fully qualified name of the specified file at the time it was opened.
3	2	Current file name length. The length in bytes of the file name returned by item 4.
4	*	Current file name. The current fully qualified name of the specified file. This might differ from item 2 because the file might have been renamed since it was opened.
5	2	DEFINE name length. For files opened with a DEFINE, the length in bytes of the DEFINE name; for other files, 0.
6	*	DEFINE name. For files opened with a DEFINE, the name of the DEFINE.
7	2	Last error. The file-system error number resulting from the last file-system operation. If *filenum* identifies an open file, the last error associated with that file number is returned. If *filenum* is -1, the last error for an operation not associated with a file number is returned. See "Considerations," earlier in this subsection.
8	2	Last-error detail. Additional information, if available, for interpreting the error reported by item 7. This value might be a file-system error number or another kind of value, depending on the operation and the primary error.
9	2	Partition in error. For partitioned files, the number of the partition associated with the error reported by item 7.
10	2	Key in error. For files with alternate keys, the specifier of the key associated with the error reported by item 7.
11	4	Next record pointer. For disk files that are not key-sequenced and not accessed with alternate key, the value of the next record pointer. This item cannot be used with the 64-bit primary-key election of the FILE_OPEN_ procedure; an attempt results in error 581. Superseded by item 201.

[1] Item code 235 is supported only on systems running H-series RVUs.

Table 5-4. FILE_GETINFOLIST_ Item Codes (page 2 of 24)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects. Items described as "Applies only to SQL/MX objects" return file-system error 2 if queried on anything that is not an SQL/MX object.

Item Code	Size (Bytes)	Description
12	4	Current record pointer. For disk files that are not key-sequenced and not accessed with alternate key, the value of the current record pointer. This item cannot be used with the 64-bit primary-key election of the FILE_OPEN_ procedure; an attempt results in error 581. Superseded by item 202.
13	2	Current key specifier. For structured disk files, the key specifier of the current key.
14	2	Current key length. For structured disk files, the length in bytes of the current key value. This item cannot be used with the 64-bit primary-key election of the FILE_OPEN_ procedure; an attempt results in error 581. Superseded by item 203.
15	*	Current key value. For structured disk files, the current key value. Item 15 is not defined for queue files. This item cannot be used with the 64-bit primary-key election of the FILE_OPEN_ procedure; an attempt results in error 581. Superseded by item 204.
16	2	Current primary-key length. For structured disk files, the length in bytes of the current primary-key value. This item cannot be used with the 64-bit primary-key election of the FILE_OPEN_ procedure; an attempt results in error 581. Superseded by item 205.
17	*	Current primary-key value. For structured disk files, the current primary-key value. This item cannot be used with the 64-bit primary-key election of the FILE_OPEN_ procedure; an attempt results in error 581. Superseded by item 206.
18	6	Tape volume. For labeled tape files, the volume serial number of the reel currently being processed.
19	2	Highest open-file number. The highest file number of any currently open file.
20	2	Next open-file number. The next file number of any open file higher than the input *filenum* value; if no higher-numbered open file exists, a value of -1 is returned.

[1] Item code 235 is supported only on systems running H-series RVUs.

Table 5-4. FILE_GETINFOLIST_ Item Codes (page 3 of 24)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects. Items described as "Applies only to SQL/MX objects" return file-system error 2 if queried on anything that is not an SQL/MX object.

Item Code	Size (Bytes)	Description
21	2	Open access mode. The access mode under which the specified file has been opened. Values are: 0 read-write 1 read only 2 write only 3 extend (supported for tape, not disk)
22	2	Open exclusion mode. The exclusion mode under which the specified file has been opened. Values are: 0 shared 1 exclusive 3 protected
23	2	Open nowait depth. The number of concurrent nowait operations permitted on the specified file, as specified when the file was opened.
24	2	Open sync depth. The sync depth or receive depth under which the specified file has been opened. For details, see FILE_OPEN_ Procedure.
25	2	Open options. The miscellaneous options under which the specified file has been opened. For details, see FILE_OPEN_ Procedure.
26	4	Operation information. For particular access methods on some devices, a value associated with the last completed I/O operation. The meaning of the value is specific to the access method. For SNAX, it is the exception response identification number.
30	2	Device type. The device type associated with the specified file.
31	2	Device subtype. The device subtype associated with the specified file.
32	2	Demountable disk. For disk volumes and disk objects, 1 if the volume is demountable; 0 otherwise.
33	2	Audited disk. For disk volumes and disk objects, 1 if the volume can support audited files; 0 otherwise.

[1] Item code 235 is supported only on systems running H-series RVUs.

Table 5-4. FILE_GETINFOLIST_ Item Codes (page 4 of 24)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects. Items described as "Applies only to SQL/MX objects" return file-system error 2 if queried on anything that is not an SQL/MX object.

Item Code	Size (Bytes)	Description
34	2	Physical record length. For disk volumes and files, the maximum transfer length of the device; for processes and $RECEIVE, 132 by convention; for other devices, a configured value that generally represents some physical limit. This is always an unsigned value representing a number of bytes.
35	4	Logical device number. For processes, -1; for other files, the number of the device supporting the specified file. For partitioned files, the number of the device supporting the specified partition is returned.
36	2	Subdevice number. The number associated with a subdevice and assigned by the device subsystem.
40	2	SQL type. For disk objects: 0 Unstructured or Enscribe file 2 SQL table 4 SQL index 5 SQL protection view 7 SQL shorthand view 11 SQL/MX table or view 12 SQL/MX index
41	2	File type. For disk objects other than SQL shorthand views: 0 unstructured 1 relative 2 entry-sequenced 3 key-sequenced
42	2	File code. For disk objects other than SQL shorthand views, the application-defined file code.
43	2	Logical record length. For structured disk objects, the maximum number of bytes in a logical record. Superseded by item 196.
44	2	Block length. For structured disk objects, the length in bytes of each block of records in the file; for unstructured disk files, the size in bytes of the system buffer used internally. Superseded by item 197.

[1] Item code 235 is supported only on systems running H-series RVUs.

Table 5-4. FILE_GETINFOLIST_ Item Codes (page 5 of 24)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects. Items described as "Applies only to SQL/MX objects" return file-system error 2 if queried on anything that is not an SQL/MX object.

Item Code	Size (Bytes)	Description
45	2	Key offset. For key-sequenced disk files, the byte offset from the beginning of the record to the primary key field. Superseded by item 198.
46	2	Key length. For key-sequenced disk files, the maximum number of bytes in the file's primary key field.
47	2	Lock key length. For key-sequenced disk files, the generic-lock key length. If this value has never been set, the key length of the file (the value of item 46) is returned. For information about generic locking, see the *Enscribe Programmer's Guide*.
48	2	Queue File. For disk objects, this is nonzero if the object is a queue file; otherwise it is zero.
50	2	Primary extent size. For disk objects other than SQL shorthand views, the size in pages (2048-byte units) of the first extent. A returned value of -1 means that the extent size cannot fit into this unsigned 2-byte attribute. Item 199 must be used to get the correct value. Superseded by item 199.
51	2	Secondary extent size. For disk objects other than SQL shorthand views, the size in pages (2048-byte units) of extents after the first extent. A returned value of -1 means that the extent size cannot fit into this unsigned 2-byte attribute. Item 200 must be used to get the correct value. Superseded by item 200.
52	2	Maximum extents. For disk objects other than SQL shorthand views, the maximum number of extents the object is allowed to have.
53	2	Allocated extents. For disk objects other than SQL shorthand views, the number of extents currently allocated for the file.
54	8	Creation time. For disk objects other than SQL shorthand views, the Julian GMT timestamp of the file's creation.
56	8	Last open time. For disk objects other than SQL shorthand views, the Julian GMT timestamp of the last time the file was opened.

[1] Item code 235 is supported only on systems running H-series RVUs.

Table 5-4. FILE_GETINFOLIST_ Item Codes (page 6 of 24)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects. Items described as "Applies only to SQL/MX objects" return file-system error 2 if queried on anything that is not an SQL/MX object.

Item Code	Size (Bytes)	Description
57	8	Expiration time. For disk objects other than SQL shorthand views, the Julian GMT timestamp giving the time before which the file cannot be purged. If this attribute has not been set, the returned field is zero-filled.
58	2	File owner. For disk objects, the user ID number that identifies the owner of the file.
59	2	Safeguard security. For disk objects, 1 if the file is under the protection of the Safeguard security system; 0 otherwise. This code is equivalent to bit 14 of code 169.
60	2	Progid security. For disk objects, 1 if a process using the file as its program file is to use the file owner's user ID as the process access ID; 0 otherwise.
61	2	Clear on purge. For disk objects, 1 if the area of disk occupied by the file should be erased (overwritten with zeros) when the file is purged; 0 otherwise.
62	4	Operating system security string. For disk objects, an array of four one-byte values specifying (from left to right) who can read, write, execute, and purge the file. Each byte contains one of these values: 0 any local ID 1 member of owner's group (local) 2 owner (local) 4 any network user (local or remote) 5 member of owner's community 6 local or remote user having same ID as owner 7 local super ID only This value is not defined if the file is under Safeguard security.
63	2	Licensed file. For disk files, 1 if the file is licensed to run in privileged mode; 0 otherwise.
65	2	Odd unstructured file. For unstructured files, 1 if I/O transfers occur with the exact byte counts specified; 0 if transfers are rounded up to an even-byte boundary.
66	2	Audited file. For disk objects other than SQL shorthand views, 1 if the object is audited by the Transaction Management Facility (TMF) subsystem; 0 otherwise.

[1] Item code 235 is supported only on systems running H-series RVUs.

Table 5-4. FILE_GETINFOLIST_ Item Codes (page 7 of 24)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects. Items described as "Applies only to SQL/MX objects" return file-system error 2 if queried on anything that is not an SQL/MX object.

Item Code	Size (Bytes)	Description
67	2	Audit compression. For disk objects other than SQL shorthand views, 1 if audit data for this file is to be compressed; 0 otherwise.
68	2	Data compression. For key-sequenced disk objects, 1 if the entries in data blocks are to be compressed; 0 otherwise.
69	2	Index compression. For key-sequenced disk objects, 1 if the entries in index blocks are to be compressed; 0 otherwise.
70	2	Refresh EOF. For disk objects other than SQL shorthand views, 1 if a change to the end-of-file value is to cause the file label to be written immediately to disk; 0 otherwise.
71	2	Create options. For disk objects, the miscellaneous attributes of the file in the form specified in the *options* parameter of FILE_CREATE_. These attributes are also available as separate items (items 65 through 70).
72	2	Write through. For disk objects, 1 if the file label specifies the use of write-through caching; 0 indicates that buffered writes are used.
73	2	Verify writes. For disk objects other than SQL shorthand views, 1 if the file label specifies that writes to the file are to be read back and the data verified; 0 otherwise.
74	2	Serial writes. For disk objects other than SQL shorthand views, 1 if the file label specifies that writes are to be made serially to the mirrors when a file resides on a mirrored disk; 0 indicates that the system can choose to do either serial or parallel writes.
75	2	File is open. For disk objects other than SQL shorthand views, 1 if the object is either open or has an incomplete TMF transaction against it; 0 otherwise. This value should always be 1 when it is obtained by a call to FILE_GETINFOLIST_.
76	2	Crash open. For disk objects other than SQL shorthand views, 1 if the object was open with write access when a system failure occurred and the object has not been opened since; 0 otherwise. This value should always be 0 when obtained by a call to FILE_GETINFOLIST_.

[1] Item code 235 is supported only on systems running H-series RVUs.

Table 5-4. FILE_GETINFOLIST_ Item Codes (page 8 of 24)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects. Items described as "Applies only to SQL/MX objects" return file-system error 2 if queried on anything that is not an SQL/MX object.

Item Code	Size (Bytes)	Description
77	2	Rollforward needed. For disk objects other than SQL shorthand views, 1 if the object has the TMF rollforward-needed flag set; 0 otherwise.
78	2	Broken. For disk objects other than SQL shorthand views, 1 if the object has the broken flag set; 0 otherwise.
79	2	Corrupt. For disk objects other than SQL shorthand views, 1 if the object has the corrupt flag set; 0 otherwise.
80	2	Secondary partition. For disk objects, 1 if the file is a secondary partition of a partitioned file.
81	2	Index levels. For key-sequenced disk objects, the number of levels currently used in the key indexing structure.
82	2	SQL program. For disk objects, 1 if the file is a program file containing compiled SQL statements; 0 otherwise.
83	2	SQL valid. For disk objects, 1 if the file is a program file containing compiled SQL statements and the compilation is probably valid.
84	2	SQL-catalog name length. For disk objects, the number of bytes in the name of the SQL catalog associated with the object; 0 if no catalog is associated with the object.
85	*	SQL-catalog name. For disk objects, the fully qualified name of the SQL catalog associated with the object. The length of the name is given by item 84.
90	2	Number of partitions. For disk objects, the number of secondary partitions the disk object has.

[1] Item code 235 is supported only on systems running H-series RVUs.

Table 5-4. FILE_GETINFOLIST_ Item Codes (page 9 of 24)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects. Items described as "Applies only to SQL/MX objects" return file-system error 2 if queried on anything that is not an SQL/MX object.

Item Code	Size (Bytes)	Description
91	*	Partition descriptors. For disk files, an array of 4-byte partition descriptors, one for each secondary partition. Each entry has this structure: `INT` *`primary-extent-size`*; `INT` *`secondary-extent-size`*; These values give the primary and secondary extent sizes in pages (2048-byte units). The length in bytes of this item is 4 times the value of item 90. A value of -1 in either of the fields means that the corresponding extent size cannot fit into the unsigned 2-byte field. Item 97 must be used to get the correct value. Superseded by item 97.
92	*	Partition-volume name-length array. For disk files, an array of INT values, each giving the length in bytes of the volume name (supplied in item 93) on which the corresponding secondary partition resides. The length in bytes of this item is 2 times the value of item 90.
93	*	Partition-volume names. For disk files, the concatenated volume names of the secondary partitions. The names are fully qualified. The length of each is given by the corresponding entry in item 92. The length of this item is given by item 96, or equivalently, by the sum of the elements in item 92.
94	2	Partition partial-key length. For partitioned key-sequenced disk files, the number of bytes of the primary key that are used to determine which partition of the file contains a particular record.
95	*	Partition partial-key values. For partitioned key-sequenced disk files, the concatenated partial-key values. Since the number of entries is given by item 90, and the size of each entry is given by item 94, the size of item 95 is the product of those two values.
96	2	Partition-volume names total length. For disk files, the total number of bytes occupied by the concatenated volume names of the secondary partitions. This is the same as the sum of the elements in item 92.

[1] Item code 235 is supported only on systems running H-series RVUs.

Table 5-4. FILE_GETINFOLIST_ Item Codes (page 10 of 24)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects. Items described as "Applies only to SQL/MX objects" return file-system error 2 if queried on anything that is not an SQL/MX object.

Item Code	Size (Bytes)	Description

97	*	Partition descriptors (32-bit). An array of 8-byte values, one for each secondary partition. Each entry has this structure:
		`INT (32) primary-extent-size;`
		`INT (32) secondary-extent-size;`
		These values give the primary and secondary extent sizes in pages. The length of this item in bytes is eight times item 90. Supersedes item 91.

98	*	Partition-volume relative names-length array. For disk files, an array of INT byte counts, each giving the length of the corresponding secondary partition-volume name in the form returned by item 99. The length of this item is two times item 90. The length of this item is the sum of the elements in item 98.
99	*	Partition-volume relative names. For disk files, the concatenated names of the volumes of the secondary partitions. Unlike item 93, the names can be missing a system name (implying the system of the primary file) depending on how the names were specified when the file was created.
100	2	Number of alternate keys. For disk files, the number of alternate-key fields.
101	*	Alternate-key descriptors. For disk files, an array of key-descriptor entries, one for each alternate key. Each entry is 12 bytes long. The structure of each entry is described under item 101 in Table 5-3 on page 5-30 (under FILE_CREATELIST_). The length in bytes of this item is 12 times the value of item 100. Superseded by item 106.
102	2	Number of alternate-key files. For disk files, the number of files holding alternate-key records.
103	*	Alternate-file name-length array. For disk files, an array of INT values, each giving the length in bytes of the corresponding alternate-file name found in item 104. The length in bytes of this item is 2 times the value of item 102.

[1] Item code 235 is supported only on systems running H-series RVUs.

Table 5-4. FILE_GETINFOLIST_ Item Codes (page 11 of 24)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects. Items described as "Applies only to SQL/MX objects" return file-system error 2 if queried on anything that is not an SQL/MX object.

Item Code	Size (Bytes)	Description
104	*	Alternate-file names. For disk files, the concatenated names of the alternate-key files. The names are fully qualified. The length of each is given by the corresponding entry in item 103. The length of this item is given by item 105, or equivalently, by the sum of the elements of item 103.
105	2	Alternate-file total name length. For disk files, the total number of bytes occupied by the concatenated alternate-file names. This is the same as the sum of the elements in item 103.

106	*	Alternate-key descriptors (32-bit). An array of 14-byte descriptor entries, one for each alternate key. Each entry has this structure:
		`INT key-specifier;`
		`INT key-len;`
		`INT (32) key-offset;`
		`INT key-filenum;`
		`INT null-value;`
		`INT attributes;`
		The *attributes* parameter has these fields:
		<0> Do not index when null.
		<1> Unique.
		<2> Do not update.
		<3> Insertion order duplicate.

108	*	Alternate-file relative name-length array. For disk files, an array of INT byte counts, each giving the length of the corresponding alternate-key file name in the form returned by item 109. The length of this item is two times item 102.
109	*	Alternate-file relative names. For disk files, the concatenated names of the alternate-key files. Unlike item 104, the system name can be left out (implying the system of the primary file) depending on how the names were specified when the file was created. The length of each name is given by the corresponding entry in item 108. The length of this item is given by the sum of the elements of item 108.

[1] Item code 235 is supported only on systems running H-series RVUs.

Table 5-4. FILE_GETINFOLIST_ Item Codes (page 12 of 24)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects. Items described as "Applies only to SQL/MX objects" return file-system error 2 if queried on anything that is not an SQL/MX object.

Item Code	Size (Bytes)	Description
110	4	Volume capacity. For disk volumes and disk objects, the data capacity of the volume as indicated in the volume label, expressed in pages (2048-byte units). This value accounts for disk space used for data protection (such as spare sectors), but not for other system uses. For files residing on Storage Management Foundation (SMF) virtual disks, this item code will return a -1.
111	4	Volume free space. For disk volumes and disk objects, the total free space currently available on the volume, in pages (2048-byte units). For files residing on Storage Management Foundation (SMF) virtual disks, this item code will return a 0.
112	4	Volume fragments. For disk volumes and disk objects, the number of individual free space fragments on the volume. For files residing on SMF (Storage Management Foundation) virtual disks, this item code will return a 0.
113	4	Largest volume fragment. For disk volumes and disk objects, the size in pages of the largest free space fragment on the volume. For files residing on SMF (Storage Management Foundation) virtual disks, this item code will return a 0.
114	16	Disk drive types. For disk volumes and disk objects, the types of drives on which the volume is mounted. This item contains two 8-byte fields that give the types of the primary and mirror drives respectively. Each field is a drive-product number in ASCII. If the information is unavailable for a drive (because it's inaccessible or not configured), the corresponding field is returned blank. For drive models 4110 and 4120, which cannot be distinguished by software, the returned value is "4110 ". For files residing on SMF (Storage Management Foundation) virtual disks, this item code will return blanks.

[1] Item code 235 is supported only on systems running H-series RVUs.

Table 5-4. FILE_GETINFOLIST_ Item Codes (page 13 of 24)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects. Items described as "Applies only to SQL/MX objects" return file-system error 2 if queried on anything that is not an SQL/MX object.

Item Code	Size (Bytes)	Description
115	8	Disk drive capacities. For disk volumes and disk objects, the capacities in pages (2048-byte units) of the primary and mirror drives on which the volume is mounted. This item contains two INT(32) values, expressing in pages (2048-byte units) the capacities of the primary and mirror drives respectively. The values account for disk space used for data protection (such as spare sectors), but not for other system uses. If the information is unavailable for a drive (because it is inaccessible or not configured), 0D is returned for that drive. For files residing on SMF (Storage Management Foundation) virtual disks, this item code will return blanks.
116	2	Sequential block buffering. 1 if the open is using a sequential block buffer; 0 otherwise.
117	8	Last open LCT. For disk objects, the timestamp of the last time the file was opened, expressed in the local civil time (LCT) of the system on which the file resides.
118	8	Expiration LCT. For disk objects, the timestamp giving the time before which the file cannot be purged, expressed in the local civil time (LCT) of the system on which the file resides. If this attribute has not been set, the returned field is zero-filled.
119	8	Creation LCT. For disk objects, the timestamp of the file's creation, expressed in the local civil time (LCT) of the system on which the file resides.
136	4	Partition EOF. For disk objects other than SQL shorthand views, the end-of-file value of the partition named in the open operation (when returned by FILE_GETINFOLIST_) or of the partition named in this call (when returned by FILE_GETINFOLISTBYNAME_). A returned value of -1 means that the end-of-file value cannot fit into this unsigned 4-byte attribute. Item 193 must be used to get the correct value. Superseded by item 193.

[1] Item code 235 is supported only on systems running H-series RVUs.

Table 5-4. FILE_GETINFOLIST_ Item Codes (page 14 of 24)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects. Items described as "Applies only to SQL/MX objects" return file-system error 2 if queried on anything that is not an SQL/MX object.

Item Code	Size (Bytes)	Description
137	4	Partition maximum size. For disk objects other than SQL shorthand views, the maximum allowable size in bytes of the partition named in the open operation (when returned by FILE_GETINFOLIST_) or of the partition named in this call (when returned by FILE_GETINFOLISTBYNAME_). A returned value of -1 means that the partition maximum size cannot fit into this unsigned 4-byte attribute. Item 194 must be used to get the correct value. Superseded by item 194.
140	8	Partition modification time. For disk objects other than SQL shorthand views, the Julian GMT timestamp indicating the last modification time of the partition named in the open operation (when returned by FILE_GETINFOLIST_) or of the partition named in this call (when returned by FILE_GETINFOLISTBYNAME_).
141	8	Partition modification LCT. For disk objects other than SQL shorthand views, the timestamp indicating the last modification time of the partition named in the open operation (when returned by FILE_GETINFOLIST_) or of the partition named in this call (when returned by FILE_GETINFOLISTBYNAME_). The time is expressed in the local civil time (LCT) of the system on which the file resides. It is derived from the Julian GMT partition modification time (item code 140).
142	4	Aggregate EOF. For disk objects, the end-of-file value of the file. For a partitioned file where the entire file has been opened, the end-of-file value of the entire file is returned. A returned value of %hFFFFFFFF indicates that the end-of-file value cannot fit into this unsigned 4-byte attribute. In this case, to obtain the end-of-file value, use the 8-byte attribute, which is item code 191. Superseded by item 191.
143	4	Aggregate maximum file size. For disk objects, the maximum allowable size in bytes of the file. For a partitioned file where the entire file has been opened, the maximum size of the entire file is returned. A returned value of %hFFFFFFFF indicates that the maximum file size cannot fit into this unsigned 4-byte attribute. In this case, to obtain the maximum file size, use the 8-byte attribute, item code 192. Superseded by item 192.

[1] Item code 235 is supported only on systems running H-series RVUs.

Table 5-4. FILE_GETINFOLIST_ Item Codes (page 15 of 24)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects. Items described as "Applies only to SQL/MX objects" return file-system error 2 if queried on anything that is not an SQL/MX object.

Item Code	Size (Bytes)	Description
144	8	Aggregate modification time. For disk objects, the Julian GMT timestamp indicating the last modification time of the file. For a partitioned file, the most recent modification time of all accessible partitions.
145	8	Aggregate modification LCT. For disk objects, the timestamp indicating the last modification time of the file, expressed in the local civil time (LCT) of the system on which the file resides. For a partitioned file, the most recent modification time of all accessible partitions.
153	2	Logical (packed) record length. In an SQL object, the maximum number of bytes in a packed record.
160	6	Three-word partition modification LCT. For disk objects other than SQL shorthand views, the three-word timestamp indicating the last modification time of the partition named in the open operation (when returned by FILE_GETINFOLIST_) or of the partition named in this call (when returned by FILE_GETINFOLISTBYNAME_). The time is expressed in the local civil time (LCT) of the system on which the file resides and represents the number of 10-millisecond units since midnight (00:00) on December 31, 1974.
161	2	OSS file. 1 if the file is an OSS file; 0 otherwise.
164	4	OSS file owner's group ID. The group ID is a number in the range 0 through 65535.
165	4	OSS access permissions. Applies only to OSS files. See the `chmod(2)` function reference page either online or in the *Open System Services System Calls Reference Manual* for a description of OSS access permissions.
166	2	OSS open. 1 if the open was performed by any of these OSS functions: `creat()`, `chdir()`, `open()`, or `opendir()`; 0 otherwise. This code applies only to FILE_GETINFOLIST_.
167	4	File owner. For disk files, the user ID number that identifies the owner of the file. The user ID is a number in the range 0 through 65535.
168	2	OSS number of links. Applies only to OSS files.

[1] Item code 235 is supported only on systems running H-series RVUs.

Table 5-4. FILE_GETINFOLIST_ Item Codes (page 16 of 24)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects. Items described as "Applies only to SQL/MX objects" return file-system error 2 if queried on anything that is not an SQL/MX object.

Item Code	Size (Bytes)	Description
169	2	Security mechanisms in effect. For disk objects, the word of bits indicates the security mechanisms in effect for the given object. Note that the security mechanism of an object is not necessarily related to the overall security of the subvolume or volume. <0:10> reserved <11> POSIX ACL file security <12> OSS security <13> SQL security <14> Safeguard file security <15> Guardian file security Note: Bit 11 is supported only on systems running G06.29 and subsequent RVUs. For systems running H-series RVUs, it is a reserved bit.
176	2	Unreclaimed free space. For SQL tables and indexes, 1 if the object has the F flag (UNRECLAIMED FREE SPACE) set; 0 otherwise.
177	2	Incomplete SQLDDL operation. For SQL tables and indexes, 1 if the object has the D flag (INCOMPLETE SQLDDL OPERATION) set; 0 otherwise.
178	2	Physical volume name length. The length in bytes of the name returned by item 179.
179	*	Physical volume name. For disk objects, the name of the volume on which the object resides, in external form with system name. This can be different from the volume indicated in the object's name (for example, for NonStop Storage Manager Foundation (SMF) objects).
180	2	Physical volume primary processor. For disk objects, the processor number that contains the current primary disk process supporting the volume on which the object resides.
182	2	Physical file name length. The length in bytes of the name returned by item 183.
183	*	Physical file name. For SMF disk objects, the name of the physical file where the data resides, in fully qualified external form. For other disk objects, this item has zero length.

[1] Item code 235 is supported only on systems running H-series RVUs.

Table 5-4. FILE_GETINFOLIST_ Item Codes (page 17 of 24)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects. Items described as "Applies only to SQL/MX objects" return file-system error 2 if queried on anything that is not an SQL/MX object.

Item Code	Size (Bytes)	Description
184	2	Referencing logical name length. The length in bytes of the name returned by item 185.
185	*	Referencing logical name. For a physical file containing the data of a SMF virtual disk object, the name of that SMF object in fully qualified external form. For other disk objects, including those specified by a SMF logical name, this item has zero length.
191	8	Aggregate EOF 64-bit. For disk objects, the end-of-file value of the file. For a partitioned file where the entire file has been opened, the end-of-file value of the entire file is returned.
192	8	Aggregate maximum file size 64-bit. For disk objects, the maximum allowable size in bytes of the file. For a partitioned file where the entire file has been opened, the maximum size of the entire file is returned.
193	8	Partition end-of-file (64-bit). For disk objects other than SQL shorthand views, the number of bytes in the object. If the file is partitioned, determined from the specified partition only.
194	8	Partition maximum size (64-bit). For disk objects other than SQL shorthand views, the maximum number of bytes the object is allowed to contain. If the file is partitioned, determined from the specified partition only.
195	2	File format. Returns the file format (1 or 2). Format 1 files allow only as many as 4 KB blocks and as many as 2 GB partitions; Format 2 files allow larger blocks and partitions.
196	4	Logical record length (32-bit). For structured disk files, the maximum number of bytes in a logical record. Supersedes item 43.
197	4	Block length (32-bit). For structured files, the size of a block of records. For unstructured files, the unstructured buffer size. Supersedes item 44.
198	4	Key offset (32-bit). For key-sequenced disk files, the byte offset from the beginning of the record to the primary-key field. Supersedes item 45.

[1] Item code 235 is supported only on systems running H-series RVUs.

Table 5-4. FILE_GETINFOLIST_ Item Codes (page 18 of 24)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects. Items described as "Applies only to SQL/MX objects" return file-system error 2 if queried on anything that is not an SQL/MX object.

Item Code	Size (Bytes)	Description
199	4	Primary extent size (32-bit). The size in pages (2048-byte units) of the first extent. Supersedes item 50.
200	4	Secondary extent size (32-bit). The size in pages (2048-byte units) of extents after the first extent. Supersedes item 51.
201	8	Next record pointer (64-bit). For opened disk files other than key-sequenced, and not accessed with alternate key, the setting of the next-record pointer in 64-bit form. Supersedes item 11.
202	8	Current record pointer (64-bit). For opened disk files other than key-sequenced, and not accessed with alternate key, the setting of the current record pointer in 64-bit form. Supersedes item 12.
203	2	Current key length. For opened structured disk files, the length in bytes of the current key value (see item 204). Supersedes item 14.
204	*	Current key value (64-bit). The current key value for opened structured disk files. The length is given by item 203. Supersedes item 15. This item differs from item 15 for non-key-sequenced files when the current key is the primary key, in which case the 64-bit form of the key is returned instead of the 32-bit form.
205	2	Current Primary-key length. For opened structured disk files, the length in bytes of the current primary-key value (obtained using item 206). Supersedes item 16.
206	*	Current Primary-key value (64-bit). The current primary-key value for opened structured disk files. The length is given by item 205. Supersedes item 17. This item differs from item 17 in that for non-key-sequenced files, the 64-bit form of the key is returned instead of the 32-bit form.
212	2	Block checksumming option. For Format 2 structured files, 1 indicates that the checksum calculation and comparison is used; 0 indicates it is not used.

[1] Item code 235 is supported only on systems running H-series RVUs.

Table 5-4. FILE_GETINFOLIST_ Item Codes (page 19 of 24)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects. Items described as "Applies only to SQL/MX objects" return file-system error 2 if queried on anything that is not an SQL/MX object.

Item Code	Size (Bytes)	Description
221	*	Partition maximum extent size array. For partitioned disk files, this item is an array of INT of the maximum extents value for each secondary partition. The length of the array is 2 times the number of partitions (item 90).
225	2	SQL/MX object. Applies only to disk objects. 1 if the object is an SQL/MX object, 0 otherwise.
226	2	SQL/MX physical object. Applies only to SQL/MX objects. <0:14> reserved <15> 1 if resource fork, 0 otherwise
227	2	MX partition method. Applies only to SQL/MX objects. 1 SQL/MX range partitioned 2 SQL/MX hash partitioned
228	2	ANSI name length. Applies only to SQL/MX objects. The length, in bytes, of the ANSI name. The length is 0 if the MX object has no ANSI name (for example, a resource fork).
229	*	ANSI name. Applies only to SQL/MX objects. The ANSI name of the MX object. The length of the name is given bsy item 228.
230	2	ANSI name space. Applies only to SQL/MX objects. The name of the ANSI name space. The value is either "TA" or "IX".
235[1]	2	Direct I/O buffer protection. Applies only to disk objects. The state of the TRUST flag indicating direct I/O access permission to user buffers when the process is running. The values are: 0 TRUST flag is disabled 1 TRUST flag is enabled for private access to the process 3 TRUST flag is enabled for shared access to the process
236	32	Disk drive types2. For disk volume and disk objects, the types of drives on which the volume is mounted. This item returns the same information as the info item 114 (Disk Drive Type) except that it returns 16 bytes for each for the primary and secondary drives. Blanks are returned if one of the drives is not present.

[1] Item code 235 is supported only on systems running H-series RVUs.

Table 5-4. FILE_GETINFOLIST_ Item Codes (page 20 of 24)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects. Items described as "Applies only to SQL/MX objects" return file-system error 2 if queried on anything that is not an SQL/MX object.

Item Code	Size (Bytes)	Description
237	10	ErrorSetExternally. For disk files, an array of 5 INT values is returned. 0 Indicates if the file-system error variables are set externally anytime for that file. If the value is 0, it implies that the file-system errors have not been overridden externally. If the value is 1, it implies that the file-system errors have been overridden externally. That is, FILE_SETLASTERROR_ has been invoked for this file. 1 Provides *last-error* value before it was overridden, otherwise 0 is returned. 2 Partition in error before overridden, otherwise 0 is returned. 3 Key in error before overridden, otherwise 0 is returned. 4 Provides *error-detail* overridden, otherwise 0 is returned. The item code returns this information for the specified *filenum*. If the *filenum* identifies an open file, the information associated with that file number is returned. If the *filenum* is -1, the information for an operation not associated with a file number (such as a purge, waited open, or failed create operation) is returned. For more information about the file-system error overrides, see FILE_SETLASTERROR_ Procedure.
1001	2	Tape process: Density -1 unsupported or unknown 0 800 bpi (NRZI) 1 1600 bpi (PE) 2 6250 bpi (GCR) 8 38000 bpi 9 Digital Data Storage (DDS)
1002	2	Tape process: Compression -1 unsupported or unknown 1 compression disabled 2 compression enabled

[1] Item code 235 is supported only on systems running H-series RVUs.

Table 5-4. FILE_GETINFOLIST_ Item Codes (page 21 of 24)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects. Items described as "Applies only to SQL/MX objects" return file-system error 2 if queried on anything that is not an SQL/MX object.

Item Code	Size (Bytes)	Description
1003	2	Tape process: Tapemode
		-1 unsupported or unknown 0 startstop 1 streaming 2 slow streaming 3 fast streaming
1004	2	Tape process: Level of buffering
		-1 unsupported or unknown 0 record 1 file 2 reel
1005	2	Tape or Open SCSI process: Device subtype
		-1 unknown 0 passthrough mode 5 5120 tape drive 6 5160 or 5170 tape drive 7 5130 tape drive 8 5180 tape drive 9 5190 tape drive 10 5188 tape drive 11 5142 tape drive 14 521A, 524A, or 525A tape drives
1006	2	Tape process: Automatic Cartridge Loader (ACL) status
		-1 not installed or unknown 1 installed
1007	2	Tape process: Number of tracks
		-1 unknown 0 not applicable 9 9-track tape drive 18 18-track tape drive 36 36-track tape drive
1008	2	Open SCSI process: SIM queue status
		0 not frozen 1 frozen

[1] Item code 235 is supported only on systems running H-series RVUs.

Table 5-4. FILE_GETINFOLIST_ Item Codes (page 22 of 24)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects. Items described as "Applies only to SQL/MX objects" return file-system error 2 if queried on anything that is not an SQL/MX object.

Item Code	Size (Bytes)	Description
1009	2	Tape or Open SCSI process: Current device state 0 up 1 down
1010	2	Tape process: current device status 0 not ready 1 online or ready
1011	2	Tape process: Short write mode -1 unsupported or unknown 0 allow writes shorter than 24 bytes; a record shorter than 24 bytes is padded with zeros to a length of 24 bytes (default). 1 disallow writes shorter than 24 bytes. 2 allow writes shorter than 24 bytes; no padding is done on records shorter than 24 bytes.
1012	2	Tape process: Checksum mode 0 normal I/O mode 1 checksum mode
1013	2	Tape or Open SCSI process: Tracing level 0 no tracing *x* *x* tracing level
1014	2	Tape or Open SCSI process: Number of openers 1 1 opener (might be exclusive) *x* *x* openers
1015	2	Tape process: Current controller state -1 unknown 0 unloaded 1 loaded
1016	2	Tape process: Current assignment status -1 no applicable 0 unassigned 1 assigned (5188 tape drive only)

[1] Item code 235 is supported only on systems running H-series RVUs.

Table 5-4. FILE_GETINFOLIST_ Item Codes (page 23 of 24)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects. Items described as "Applies only to SQL/MX objects" return file-system error 2 if queried on anything that is not an SQL/MX object.

Item Code	Size (Bytes)	Description
1017	2	Tape process: Current tape movement -1 not at BOT 0 at BOT
1018	2	Tape process: Return end-of-tape (EOT) message when writing labeled tapes 0 volume switching is transparent 1 notify user of volume switch by sending error 150 (EOT). COBOL applications do not receive error 150 (EOT); the COBOL run-time library (RTL) handles this error transparently.
1019	2	Tape process: Pending errors
1020	2	Tape process: Reason for downing
1021	2	Tape process: Current checkpoint state
1022	2	Open SCSI process: Number of open paths
1023	2	Open SCSI process: Maximum number of I/O requests
1024	2	Open SCSI process: Number of pending I/Os
1025	2	Open SCSI process: Highest number of pending I/Os since the IOP was started
1028	2	Open SCSI process: Maximum number of openers
1900	4	Tape or Open SCSI process: Maximum transfer length allowed 32767 maximum transfer length is 32767 bytes 57344 maximum transfer length is 57344 bytes
3102	2	Tape process: Length in bytes of attribute 3103. 0 if no tape is mounted or the tape does not have labels.

[1] Item code 235 is supported only on systems running H-series RVUs.

Table 5-4. FILE_GETINFOLIST_ Item Codes (page 24 of 24)

Items in this table with a size of 2 bytes are of data type INT. The term "disk file" applies only to Enscribe files. The term "disk object" applies to Enscribe files and SQL objects. Items described as "Applies only to SQL/MX objects" return file-system error 2 if queried on anything that is not an SQL/MX object.

Item Code	Size (Bytes)	Description
3103	up to 240	Tape process: Automatic Volume Recognition (AVR) labels Beginning-of-volume label (VOL1) label and the first beginning-of-file-section label group (HDR1, HDR2).
3104	2	Tape process: Length in bytes of attribute 3105. 0 if no tape is mounted or the tape does not have labels.
3105	up to 160	Tape process: Current labels Beginning-of-file-section label group (HDR1, HDR2) of the current file.

[1] Item code 235 is supported only on systems running H-series RVUs.

OSS Considerations

- These item codes are not applicable to OSS objects but do not cause an error to be returned: 50, 51, 59, 60, 61, 62, 192, 199, and 200.

Example

```
itemlist := 3;      ! return current file name length
itemlist[1] := 4;   ! return current file name
error := FILE_GETINFOLIST_ ( filenumber, itemlist, 2,
                             result^buffer,
                             result^max );
```

Related Programming Manuals

For programming information about the FILE_GETINFOLIST_ procedure, see the *Guardian Programmer's Guide*. For information on the SQL objects and programs, see the *HP NonStop SQL/MP Programming Manual for C* and the *HP NonStop SQL/MP Programming Manual for COBOL*.

FILE_GETINFOLISTBYNAME_ Procedure

Summary

The FILE_GETINFOLISTBYNAME_ procedure obtains detailed information about a file identified by file name.

A related (and simpler to use) procedure, FILE_GETINFOBYNAME_, obtains a limited set of information about a file identified by file name.

Syntax for C Programmers

Note. In the TNS/E environment the CEXTDECS file uses the *int* data type for 32-bit values. This is a change from the TNS and TNS/R environments where CEXTDECS uses the *long* data type for 32-bit values.

```
#include <cextdecs(FILE_GETINFOLISTBYNAME_)>

short FILE_GETINFOLISTBYNAME_ ( const char *filename
                               ,short length
                               ,short *item-list
                               ,short number-of-items
                               ,short *result
                               ,short result-max
                               ,[ short *result-length ]
                               ,[ short *error-item ] );
```

Syntax for TAL Programmers

```
error := FILE_GETINFOLISTBYNAME_ ( filename:length     ! i:i
                                  ,item-list           ! i
                                  ,number-of-items     ! i
                                  ,result              ! o
                                  ,result-max          ! i
                                  ,[ result-length ]   ! o
                                  ,[ error-item ] );   ! o
```

Parameters

error returned value

INT

is a file-system error number indicating the outcome of the operation.

filename:*length* input:input

STRING .EXT:ref:*, INT:value

specifies the Guardian name of the file of interest. The value of *filename* must be exactly *length* bytes long and must be a valid file name or DEFINE name. If the name is partially qualified, it is resolved using the contents of the =_DEFAULTS DEFINE.

item-list input

INT .EXT:ref.*

is an array of values that specify the items of information to be returned by the procedure. Each element of the array must be of type INT and contain a code value of 30 or greater from Table 5-4 on page 5-50 (under FILE_GETINFOLIST_).

number-of-items input

INT:value

specifies the number of items supplied in *item-list*.

result output

INT .EXT:ref:*

is the buffer in which the requested items of information are returned. The item values are returned in the order specified in *item-list*. Each item begins on an INT boundary. Every variable-length item has an associated item giving its length; the caller should put this associated item into *item-list* immediately before the variable-length item.

result-max input

INT:value

specifies the maximum size in bytes of the array of values that can be returned in *result*. If the specified size is not large enough to hold the requested items, an *error* value of 563 (buffer too small) is returned and the contents of *result* are undefined.

result-length output

INT .EXT:ref:1

returns the length in bytes of the array of values returned in *result*. *result-length* is an odd value only if the last value in the array has an odd length.

error-item output

INT .EXT:ref:1

returns the index of the item that was being processed when an error was detected. The index of the first item in *item-list* is 0.

Considerations

- Normally if an error is returned, the contents of the *result* parameter are undefined. However, if the returned error code is 2 (operation invalid for file type), the *result* parameter contains a combination of correct values (for valid items) and unchanged memory locations (for invalid items because of the kind of file). The *error-item* value points to the first invalid item.

  When error 2 occurs, any items prior to the one pointed to by *error-item* are returned with correct values in the *result* parameter; following items might or might not be valid. If a following item is known to be valid because of the kind of file involved, the correct *result* value for the item can be accessed in the corresponding location in the *result* buffer. To do so, the program will have to account for space in the buffer reserved for preceding invalid items as well as for space for preceding valid items. (Preceding in this case refers to *some* item that occurs before the item in question.) See description of items in Table 5-4 on page 5-66 to determine the kinds of files for which an item is valid.

  Invalid items that are fixed-size will have the amount of space reserved in the result buffer, but that section of buffer will be unchanged. Invalid items that are variable-sized have no space reserved for them, but this should not be depended upon because they could become valid in a future RVU and thus start occupying space. The programmer might want to place all the items that could cause error 2 in the item list after those that are not expected to cause this error.

- Specifying a subtype 30 process

  When FILE_GETINFOLISTBYNAME_ is called with a file name that designates a subtype 30 process, the procedure sends a device inquiry system message to the process to determine the device type and subtype. The message sent by FILE_GETINFOLISTBYNAME_ is in C-series format (message -40) or D-series format (message -106) depending on the options used when the subtype 30 process opened $RECEIVE through the FILE_OPEN_ procedure. For the formats of messages -40 and -106, see the *Guardian Procedure Errors and Messages Manual*.

The subtype 30 process replies with the requested information in system message -40 or -106, corresponding to the original message. The returned device type value should be one of those listed in Appendix A, Device Types and Subtypes. If the message response is incorrectly formatted, the FILE_GETINFOLISTBYNAME_ caller receives device type and subtype values of 0. The REPLY caller (the subtype 30 process) receives an error 2.

A deadlock occurs if a subtype 30 process calls FILE_GETINFOLISTBYNAME_ on its own process name.

- Last modification times and last open times

  For all items in Table 5-4 on page 5-66 that return some form of last modification time, creation time is returned for an object that has never been modified. Similarly, for items that return some form of last open time, creation time is returned for an object that has never been opened.

- Specifying a SMF logical file

  When the FILE_GETINFOLISTBYNAME_ procedure is called with a file name that designates an SMF logical file and the physical volume containing the associated physical file is inaccessible, an error is returned. An exception to this is when a call requests only items 182 and 183; in that case, the requested physical file name is returned without error, provided that the SMF virtual volume process is accessible and encounters no error.

- Secondary partition of non-SQL Enscribe files

  If the *filename* argument contains a secondary partition of an Enscribe file other than SQL, it returns only the value for the partition named. Only the primary partition of a file other than an SQL file has information on the other partitions in its label. Only when *filename* contains the primary partition do the aggregate items return the expected information.

- Support for SQL files includes both Format 1 and Format 2 files.

- Referencing Enscribe format 2 files with extent size greater than 65535 or OSS files larger than approximately 2 gigabytes.

  If the file being referenced is an Enscribe format 2 file and the extent size exceeds 65535 or OSS files larger than approximately 2 gigabytes, item codes will return -1 with no error indication.

OSS Considerations

- These item codes are not applicable to OSS objects but do not cause an error to be returned: 50, 51, 59, 60, 61, 62, 192, 199, and 200.

Example

```
itemlist := 54;        ! return creation timestamp of file
number^of^items := 1;
```

```
result^max := 8;       ! timestamp is 8 bytes long
error := FILE_GETINFOLISTBYNAME_ ( name:length, itemlist,
                                   number^of^items, result,
                                   result^max );
```

Related Programming Manuals

For programming information about the FILE_GETINFOLISTBYNAME_ procedure, see the *Guardian Programmer's Guide*. For information on the SQL objects and programs, see the *HP NonStop SQL/MP Programming Manual for C* and the *HP NonStop SQL/MP Programming Manual for COBOL*.

FILE_GETLOCKINFO_ Procedure

Summary

The FILE_GETLOCKINFO_ procedure obtains information about locks (held or pending) on a local disk file or on a set of files on a local disk volume. Each call returns information about one lock and as many holders or waiters as permitted by the caller's request. A succession of calls can obtain information about all the locks on a file or volume, or all the locks owned by a process or transaction.

Syntax for C Programmers

```
#include <cextdecs(FILE_GETLOCKINFO_)>

short FILE_GETLOCKINFO_ ( const char *name
                         ,short length
                         ,[ short *processhandle ]
                         ,[ short *transid ]
                         ,short *control
                         ,short *lock-descr
                         ,short lock-descr-length
                         ,short *participants
                         ,short max-participants
                         ,[ char *locked-name ]
                         ,[ short maxlen ]
                         ,[ short *locked-name-length ] );
```

- The parameter *maxlen* specifies the maximum length in bytes of the character string pointed to by *locked-name*, the actual length of which is returned by

locked-name-length. All three of these parameters must either be supplied or be absent.

Syntax for TAL Programmers

```
error := FILE_GETLOCKINFO_ ( name:length                  ! i:i
                            ,[ processhandle ]            ! i
                            ,[ transid ]                  ! i
                            ,control                      ! i,o
                            ,lock-descr                   ! o
                            ,lock-descr-length            ! i
                            ,participants                 ! o
                            ,max-participants             ! i
                            ,[ locked-name:maxlen ]       ! o:i
                            ,[ locked-name-length ] );    ! o
```

Parameters

error returned value

INT

is a file-system error number indicating the outcome of the operation. Possible *error* values include:

0 Information for one locked file and all its lock holders/waiters was returned without error. More locks might exist; continue calling FILE_GETLOCKINFO_.

1 End of information about locks associated with a process or transid.

2 The operation specified is not allowed on this type of file.

11 Lock information for the file, process, or transaction was not found. If any information has been returned already, it is now invalid.

12 The disk-process lock tables were changed between calls, so any previously returned information might be invalid. To start over, set *control* to 0 and call FILE_GETLOCKINFO_ again.

21 Invalid value specified for max-participants.

41 Checksum error on *control*. The *control* parameter has been altered between calls to FILE_GETLOCKINFO_ or was not initialized before the first call.

45 Information for one locked record or file has been returned, but the *participants* buffer was too small to hold all available information on lock holders/waiters. More locks might exist, so continue calling FILE_GETLOCKINFO_ (with *control* unchanged).

name:*length* input:input

STRING .EXT:ref:*, INT:value

specifies the name of the disk file or volume for which lock information is to be retrieved. The value of *name* must be exactly *length* bytes long and must be a valid disk file name or volume name; if *processhandle* or *transid* are specified, it must be a volume name. If the supplied name is partially qualified, it is resolved using the contents of the =_DEFAULTS DEFINE. The specified disk file or volume must be on the local system. The value of *name* cannot be a DEFINE name.

processhandle input

INT .EXT:ref:10

if present and not null, is the process handle of the process that is holding or waiting for the locks about which information is to be returned. A nonnull value of *processhandle* cannot be supplied when *transid* is supplied. A null process handle has -1 in each word.

transid input

INT .EXT:ref:4

if present and not null, is the transaction identifier of the transaction that is holding or waiting for the locks about which information is to be returned. A nonnull value of *transid* cannot be supplied when *processhandle* is supplied. A null transid contains 0 in each word.

control input, output

INT .EXT:ref:10

is an array of words used by FILE_GETLOCKINFO_ to control a succession of calls to the procedure. When making the first of a series of calls, you must initialize word 0 of this array to the value 0. On subsequent calls, pass the value of *control* that was returned by the previous call.

lock-descr output

INT .EXT:ref:*

points to a buffer that, on return, contains a block of words describing a lock. *lock-descr-length* specifies the size in bytes of the buffer. If the buffer is not large enough to contain the information, an *error* value of 563 (buffer too small) is returned and the contents of *lock-descr* are undefined.

The information is returned in this format:

[0]	Lock type. 0 indicates a file lock; 1 indicates a record lock.
[1]	Flags. The bits are:
	<0> = 1 Indicates a generic lock
	<1:15> (reserved)
[2]	The number of participants (the number of holders and waiters for the lock).
[3:4]	Record specifier (4 bytes) if the lock is a record lock on a Format 1 file that is not key-sequenced; undefined otherwise.
[5]	The length in bytes of the key if the lock is a record lock on either a key-sequenced file or a Format 2 non-key-sequenced file (the length is always 8 in the latter case); 0 otherwise.
[6:N]	The key value if the lock is a record lock on a key-sequenced file, or a Record specifier (8 bytes) if the lock is a record lock on a Format 2 non-key-sequenced file.

lock-descr-length input

INT:value

specifies the length in bytes of the buffer pointed to by *lock-descr*.

participants output

INT .EXT:ref:*

returns an array in which each entry describes a process or transaction that is holding or waiting for the lock described by *lock-descr*. The maximum number of processes or transactions that can be described is specified by *max-participants*. Each entry is 12 words long and has this format:

[0]	Flags. The bits have these meanings:	
<0>	= 1	The participant is identified by process handle.
	0	The participant is identified by transid.
<1:3>	= 1	The lock is granted.
	0	The lock is in the waiting state.
<4>	= 1	The lock is an intent lock internally set by DP2.
<5:11>		(Reserved)
<12:15>	Lock States. These lock states have these meanings:	
LITERAL LK^IS = 1	Intent share for file locks needed.	

LITERAL LK^IX = 2Intent exclusive for file locks. Needed to lock a record using LK^UX, LK^X.

LITERAL LK^R = 3 Used only to test for existence of KS.Record range locks (LK^S AND LK^X).

LITERAL LK^US = 4 Share for unique record locks. For KS record lock a range is not locked.

LITERAL LK^S = 5 Share for file and record locks. For KS record lock a range is locked. Lines deleted.

LITERAL LKSIX = 6 Share and intent exclusive derived state for file locks.

LITERAL LK^UX = 7 Exclusive unique record locks. For KS record lock a range is locked. Lines deleted.

LITERAL LK^X = 8 Exclusive for file and record locks. For KS record lock a range is locked.

`[1]` Reserved

`[2:11]` The process handle of the participant (if *`participants`*`[0].<0>` = 1).

`[2:5]` The transid of the participant (if *`participants`*`[0].<0>` = 0).

`max-participants` input

INT:value

specifies the maximum number of lock holders and waiters that can be described in the *`participants`* buffer.

`locked-name``:`*`maxlen`* output:input

STRING .EXT:ref:*, INT:value

if present and a volume name is supplied for *`name`*, returns the subvolume and file identifier (the two rightmost parts of the file name) of the file on which the lock is set. *`maxlen`* gives the length in bytes of the string variable *`locked-name`*.

`locked-name-length` output

INT .EXT:ref:1

returns the length in bytes of the value of *`locked-name`*.

Considerations

- The FILE_GETLOCKINFO_ procedure supports single SMF logical files but does not support entire SMF virtual volumes. If the name of an SMF logical file is supplied to this procedure, the system queries the disk process of the appropriate physical volume to obtain information about current lock holders and lock waiters

on the file. If the name of an SMF virtual volume is supplied, but not a full logical file name, an error is returned.

If you call the FILE_GETLOCKINFO_ procedure and supply the name of a physical volume, lock information is returned for any file on that volume that is open under an SMF logical file name, but the returned file name is that of the physical file supporting the logical file.

- A valid range for *max-participants* is 1-2548.

OSS Considerations

- This procedure operates only on Guardian objects. OSS files cannot have Guardian locks, so there is no information to be returned. If an OSS file is specified, error 0, indicating no error, is returned.

Example

```
! The following code obtains all the available information
! about locks on the specified disk file and about all the
! holders/waiters.

control := 0;
DO
   BEGIN
   error := FILE_GETLOCKINFO_ ( myfile:length, , , control,
                                lock^descriptor,
                                lock^descriptor^len,
                                participants,
                                max^participants );
   IF (error = 0)  ! success, but maybe more locks !  OR
      (error = 45) ! more information available !  THEN
      BEGIN
      --  process the obtained information
      END;
   END;
UNTIL (error <> 0) AND (error <> 45);

IF error <> 1 THEN     ! error 1 means end of info
   BEGIN
   -- handle the error
   END;
```

FILE_GETOPENINFO_ Procedure

Summary

The FILE_GETOPENINFO_ procedure obtains information about the opens of one disk file or all the files on a disk device, or the opens of certain nondisk devices. Each call returns information about one open; make successive calls to FILE_GETOPENINFO_ to learn about all the opens.

Syntax for C Programmers

```
#include <cextdecs(FILE_GETOPENINFO_)>

short FILE_GETOPENINFO_ ( const char *searchname
                         ,short length
                         ,long long *prevtag
                         ,[ short *primary-opener ]
                         ,[ short *backup-opener ]
                         ,[ short *accessmode ]
                         ,[ short *exclusion ]
                         ,[ short *syncdepth ]
                         ,[ char *filename ]
                         ,[ short maxlen ]
                         ,[ short *filenamelen ]
                         ,[ short *accessid ]
                         ,[ short *validmask ] );
```

- The parameter *maxlen* specifies the maximum length in bytes of the character string pointed to by *filename*, the actual length of which is returned by *filenamelen*. All three of these parameters must either be supplied or be absent.

Syntax for TAL Programmers

```
error := FILE_GETOPENINFO_ ( searchname:length       ! i:i
                            ,prevtag                 ! i,o
                            ,[ primary-opener ]      ! o
                            ,[ backup-opener ]       ! o
                            ,[ accessmode ]          ! o
                            ,[ exclusion ]           ! o
                            ,[ syncdepth ]           ! o
                            ,[ filename:maxlen ]     ! o:i
                            ,[ filenamelen ]         ! o
                            ,[ accessid ]            ! o
                            ,[ validmask ] );        ! o
```

Parameters

error returned value

INT

is a file-system error number indicating the outcome of the operation. Error 1 (EOF) indicates that there are no more opens. Error 2 (invalid operation) is returned for nondisk devices that cannot return any valid information.

searchname:*length* input:input

STRING .EXT:ref:*, INT:value

specifies the name of the disk file, volume, device, or subdevice about which open information is to be returned. The value of *searchname* cannot be a DEFINE name. If the name is partially qualified, it is resolved using the =_DEFAULTS DEFINE.

prevtag input, output

FIXED .EXT:ref:1

is a value identifying the open that was last returned. Before the first call, initialize *prevtag* to 0; on subsequent calls, pass the parameter unchanged.

primary-opener output

INT .EXT:ref:10

is the process handle of the (primary) process that has the file open.

backup-opener output

INT .EXT:ref:10

is the process handle of the backup opener process associated with the primary open. A null process handle (-1 in each word) is returned if there is no backup opener.

accessmode output

INT .EXT:ref:1

is the access mode with which the file is open. The values are:

0 read-write
1 read only
2 write only

exclusion output

INT .EXT:ref:1

is the exclusion mode with which the file is open. The values are:

0 shared
1 exclusive
2 process exclusive (supported only for Optical Storage Facility)
3 protected

syncdepth output

INT .EXT:ref:1

is the sync depth that was specified when the file was opened.

filename:*maxlen* output:input

STRING .EXT:ref:*, INT:value

returns the fully qualified file name of the file about which information is being returned. *maxlen* is the length in bytes of the string variable *filename*.

filenamelen output

INT .EXT:ref:1

is the length in bytes of the name returned in *filename*.

accessid output

INT .EXT:ref:1

is the process access ID (user ID) of the opener at the time the open was done.

validmask output

INT .EXT:ref:1

returns a value indicating which of the output parameters has returned valid information. Each parameter has a corresponding bit that is set to 1 if the parameter is valid for the device, as follows:

<0> *primary-opener*
<1> *backup-opener*
<2> *accessmode*
<3> *exclusion*
<4> *syncdepth*
<5> *filename*
<6> *accessid*

Considerations

- Opens are not returned in any defined order. In particular, when retrieving information about all opens on a disk volume, the opens for any one file might not be grouped together in the sequence of calls.
- The FILE_GETOPENINFO_ procedure supports single SMF logical files but does not support entire SMF virtual volumes. If the name of an SMF logical file is supplied to this procedure, the system queries the disk process of the appropriate physical volume to obtain information about current openers. If the name of an SMF virtual volume is supplied, but not a full logical file name, an error is returned.

  If you call the FILE_GETOPENINFO_ procedure and supply the name of a physical volume that has an open that was made on an SMF logical file name, information about the open is returned, but the returned file name is that of the physical file supporting the logical file.

Example

```
! The following code causes the names of all open files and
! the process handles of the primary and backup openers to be
! returned for the volume identified by search^name:length.

tag := 0;
DO
   BEGIN
   error := FILE_GETOPENINFO_ ( search^name:length, tag,
                                pri^opener, back^opener,,,,
                                name:max^namelen );
   END;
UNTIL error <> 0;    ! error 0 means success & more opens !
                     !  left; call again !
IF error <> 1 THEN   ! error 1 means no more opens !
   BEGIN
   -- handle error
   END;
```

FILE_GETRECEIVEINFO[L]_ Procedure

Summary

The FILE_GETRECEIVEINFO[L]_ procedure returns information about the last message read on the $RECEIVE file. Because this information is contained in the file's main-memory resident access control block (ACB), the application process is not suspended by a call to FILE_GETRECEIVEINFO[L]_. Use the FILE_GETRECEIVEINFOL_ procedure to get an information on the SERVERCLASS_SENDL_ messages larger than 32K.

Syntax for C Programmers

```
#include <cextdecs(FILE_GETRECEIVEINFO_)>

short FILE_GETRECEIVEINFO_ ( short *receive-info );


#include <cextdecs(FILE_GETRECEIVEINFOL_)>

short FILE_GETRECEIVEINFOL_ ( short _far *receive-info2 );
```

Note. To ensure that you receive valid information about the last message, call FILE_GETRECEIVEINFO[L]_ before you perform another readupdate operation on $RECEIVE.

Syntax for TAL Programmers

```
error := FILE_GETRECEIVEINFO_ ( receive-info            !o
                              ,[ dialog-info ] );       !o


error := FILE_GETRECEIVEINFOL_ ( receive-info2  );      !o
```

Parameters

error returned value

INT

is a file-system error number indicating the outcome of the operation. The only possible errors are error 16 (file not open) and program errors such as error 22 (bounds error).

0 (FEOK)

indicates a successful operation.

receive-info output

INT .EXT:ref:17 (Use with FILE_GETRECEIVEINFO_)

is a block of words describing the last message read on the $RECEIVE file. It has this structure:

[0] I/O type. Indicates the data operation last performed by the message sender. Values are:

0 Not a data message (system message)
1 Sender called WRITE
2 Sender called READ
3 Sender called WRITEREAD

[1] Maximum reply count. The maximum number of bytes of data that can be returned by REPLY (as determined by the read count of the sender).

[2] Message tag. The value that identifies the request message just read. To associate a reply with a request, the message tag is passed to the REPLY procedure. The value returned here is an integer between zero and *receive depth* - 1, inclusive, that had not been in use as a message tag. When a reply is made, its associated message tag value is made available for use as a message tag for a subsequent request message.

[3] File number. The value that identifies the file associated with this message in the requesting process. If the received message is a system message that is not associated with a specific file open, this field contains -1.

[4:5] Sync ID. The sync ID associated with this message. If the received message is a system message, this field is valid only if the message is associated with a specific file open; otherwise this field is not applicable and should be ignored. See “Considerations.”

[6:15] Sender process handle. The process handle of the process that sent the last message. For system messages other than the open, close, CONTROL, SETMODE, SETPARAM, RESETSYNC, or CONTROLBUF messages, the null process handle (-1 in each word) is returned.

[16] Open label. The value assigned by the application (when replying to the open system message) to the open on which the received message was sent. It is often used to find the open table entry for the message. If this value is unavailable, -1 is returned.

dialog-info output

INT .EXT:ref:1

is used by context-sensitive Pathway servers. The *dialog-info* parameter returns dialog information about the server-class send operation that was initiated by a requester with a Pathsend procedure call. The bits of *dialog-info* have these meanings:

[0:11] Reserved

[12:13] Dialog status. Indicates the last operation performed by the message sender. Values are:

0 Context-free server-class send operation.

1 First server-class send operation in a new dialog.

2 Server-class send operation in an existing dialog.

3 Aborted dialog. No further server-class send operations will be received in this dialog. There is no buffer associated with this value.

[14] A copy of the *flags*.<14> parameter bit in the requester's call to the Pathsend SERVERCLASS_DIALOG_BEGIN_ procedure. This bit identifies the transaction model the requester is using for dialogs. The server can abort the dialog, by replying with FEEOF, to enforce a level of transaction control that the requester has not specified.

[15] Reserved

receive-info2 output

INT .EXT:ref:* (Use with FILE_GETRECEIVEINFOL_)

is a block of 19 words describing the last message read on the $RECEIVE file. It has this structure:

[0] I/O type. Indicates the data operation last performed by the message sender. Values are:

0 Not a data message (system message)
1 Sender called WRITE
2 Sender called READ
3 Sender called WRITEREAD

[1] File number. The value that identifies the file associated with this message in the requesting process. If the received message is a system message that is not associated with a specific file open, this field contains -1.

[2] Message tag. The value that identifies the request message just read. To associate a reply with a request, the message tag is passed to the REPLY procedure. The value returned here is an integer between zero and

receive depth - 1, inclusive, that had not been in use as a message tag. When a reply is made, its associated message tag value is made available for use as a message tag for a subsequent request message.

[3] Open label. The value assigned by the application (when replying to the open system message) to the open on which the received message was sent. It is often used to find the open table entry for the message. If this value is unavailable, -1 is returned.

[4:5] Maximum reply count. The maximum number of bytes of data that can be returned by REPLY. (as determined by the read count of the sender).

[6:7] Sync ID. The sync ID associated with this message. If the received message is a system message, this field is valid only if the message is associated with a specific file open; otherwise this field is not applicable and should be ignored. See “Considerations.”

[8:17] Sender process handle. The process handle of the process that sent the last message. For system messages other than the open, close, CONTROL, SETMODE, SETPARAM, RESETSYNC, or CONTROLBUF messages, the null process handle (-1 in each word) is returned.

[18] dialog-info. It is 0 if no dialog is active. For information about dialog-info, see the *dialog-info* parameter of the FILE_GETRECEIVEINFO_ procedure on page 5-107.

Considerations

- Sync ID definition

  A sync ID is a doubleword, unsigned integer. Each opened process has its own sync ID. Sync IDs are not part of the message data; rather, the receiver of a message obtains the sync ID value associated with a particular message by calling FILE_GETRECEIVEINFO[L]_. A file’s sync ID is set to 0 when the file is opened and when the RESETSYNC procedure is called for that file (RESETSYNC can be called directly or indirectly through the CHECKMONITOR procedure).

  When a request is sent to a process (that is, when a process is the object of a CONTROL, CONTROLBUF, SETMODE, SETPARAM, open, close, read, write, or WRITEREAD operation), the system increments the requester’s sync ID just before sending the message. (Therefore, a process’s first sync ID subsequent to an open has a value of 0.)

- Duplicate requests

  The sync ID allows the server process (that is, the process reading $RECEIVE) to detect duplicate requests from requester processes. Such duplicate requests are

caused by a backup requester process reexecuting the latest request of a failed primary requester process, or by certain network failures.

Note. Neither a cancelreq operation nor an `awaitio[x]` timeout completion have any affect on the sync ID (that is, the sync ID is an ever-increasing value).

Also, the sync ID is independent of the sync depth value specified to FILE_OPEN_ or OPEN.

- Open labels

  The open label (*`receive-info`*`[16]`) allows the server to quickly find an open-table entry without having to search for it. The returned value is the same as that which the server assigned (when replying to the open message) to the open on which the received message was sent.

- Server process identifying separate opens by the same requester

  The file number (*`receive-info`*`[3]`) is used by a server process to identify separate opens by the same requester process. The returned file number value is the same as the file number used by the requester to make this request.

Example

```
error := FILE_GETRECEIVEINFO_ ( receive^info );
```

Related Programming Manual

For programming information about the FILE_GETRECEIVEINFO_ procedure, see the *Guardian Programmer's Guide*.

Note. The FILE_GETRECEIVEINFOL_ procedure is supported on systems running J06.07 and later J-series RVUs and H06.18 and later H-series RVUs.

FILE_GETSYNCINFO_ Procedure

Summary

The FILE_GETSYNCINFO_ procedure is called by the primary process of a process pair before starting a series of write operations to a file open with paired access. FILE_GETSYNCINFO_ returns a file's synchronization block so that it can be sent to the backup process in a checkpoint message.The FILE_GETSYNCINFO_ procedure supersedes the GETSYNCINFO Procedure (Superseded by FILE_GETSYNCINFO_

Procedure). Unlike the GETSYNCINFO procedure, this procedure can be used with Enscribe format 2 files and OSS files greater than approximately 2 gigabytes as well as with other files.

Note. Typically, FILE_GETSYNCINFO_ is not called directly by application programs. Instead, it is called indirectly by CHECKPOINT.

Syntax for C Programmers

```
#include <cextdecs(FILE_GETSYNCINFO_)>

short FILE_GETSYNCINFO_ (  short filenum
                         , short * infobuf
                         , short infosize );
```

Syntax for TAL Programmers

```
error := FILE_GETSYNCINFO_ ( filenum                    ! i
                           ,infobuf                     ! o
                           ,infomax                     ! i
                           ,infosize );                 ! o
```

Parameters

error

INT:value

is a file-system error code that gives the status of the operation.

filenum input

INT:value

is the number that identifies the open file.

infobuf output

INT:EXT.ref:*

is where the synchronization information is stored. The size is given in the *infomax* parameter.

infomax input

INT:value

specifies the size in bytes of the *infobuf* parameter. See “Considerations.”

infosize output

INT:EXT.ref:1

returns the size in bytes of the information stored in the *infobuf* parameter.

Considerations

- The size of the *infomax* parameter must meet these limits:
  - For nondisk files, except the Transaction Monitoring Facility (TMF) transaction pseudofile (TFILE), the size must be at least 10 bytes. For the TMF product, the size must be at least 30 bytes.
  - For disk files, the size must be at least 44 bytes for non-key-sequenced files, and the sum of the primary-key length and 44 bytes for key-sequenced files.
  - For files with alternate keys, the size must be at least primary-key length plus maximum alternate-key length and 44 bytes. The primary-key length is 8 bytes for non-key-sequenced files, and the maximum alternate-key length is the maximum value of all the alternate keys for the file.
  - For any currently supported file type, an *infomax* value of 300 is adequate.

FILE_OPEN_ Procedure

Summary

The FILE_OPEN_ procedure establishes a communication path between an application process and a file. When FILE_OPEN_ successfully completes, it returns a

file number to the caller. The file number identifies this access path to the file in subsequent file-system calls.

Syntax for C Programmers

```
#include <cextdecs(FILE_OPEN_)>

short FILE_OPEN_ ( { const char *filename |
                     const char *pathname }
                  ,short length
                  ,short *filenum
                  ,[ short access ]
                  ,[ short exclusion  ]
                  ,[ short nowait-depth ]
                  ,[ short sync-or-receive-depth ]
                  ,[ short options ]
                  ,[ short seq-block-buffer-id ]
                  ,[ short seq-block-buffer-len ]
                  ,[ short *primary-processhandle ]
                   ,[ __int32_t elections ] );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
error := FILE_OPEN_ ( {filename|pathname}:length }    ! i:i
                     ,filenum                         ! i,o
                     ,[ access ]                      ! i
                     ,[ exclusion ]                   ! i
                     ,[ nowait-depth ]                ! i
                     ,[ sync-or-receive-depth ]       ! i
                     ,[ options ]                     ! i
                     ,[ seq-block-buffer-id ]         ! i
                     ,[ seq-block-buffer-len ]        ! i
                     ,[ primary-processhandle ]       ! i
                     ,[ elections ] );                ! i
```

Parameters

error returned value

INT

is a file-system error number indicating the outcome of the operation. Some values are warnings (that is, they indicate conditions that do not prevent the file from being opened); see the *filenum* parameter, below, for determining whether the file was opened successfully.

filename:*length* input:input

STRING .EXT:ref:*, INT:value

specifies the name of the Guardian file to be opened. The value of *filename* must be exactly *length* bytes long and must be a valid file name or DEFINE name. If the name is partially qualified, it is resolved using the contents of the =_DEFAULTS DEFINE.

pathname input

specifies the OSS file to be opened. *length* is ignored; the *pathname* parameter is terminated by a null character. *options*.<10> must be set to 1 to open an OSS file by its pathname. See Appendix D, File Names and Process Identifiers, for a description of OSS pathname syntax.

filenum input, output

INT .EXT:ref:1

returns a number that is used to identify the file in subsequent file-system calls. If the file cannot be opened, a value of -1 is returned.

filenum is used as an input parameter only when you are attempting a backup open. In that case, you must supply the *primary-processhandle* parameter or else the input value of *filenum* is ignored. For a backup open, *filenum* must be the *filenum* value that was returned when the file was opened by the primary process. If a backup open is successful, the input value of *filenum* is returned unless *options*.<3> is specified, in which case a new file number is assigned for the backup open. If the backup open is unsuccessful, -1 is returned.

access input

INT:value

specifies the desired access mode of the file to be opened. (See "Considerations.") Valid values are:

0 Read-write
1 Read only
2 Write only
3 Extend (supported only for tape)

The default is 0.

exclusion input

INT:value

specifies the desired mode of compatibility with other openers of the file. (See "Considerations.") Valid values are:

0 Shared
1 Exclusive

2 Process exclusive
3 Protected

The default is 0.

nowait-depth input

INT:value

specifies whether I/O operations are to be nowait. If present and not 0, this parameter specifies the number of nowait I/O operations that can be in progress for the file concurrently with other processing. The maximum value is 1 for disk files and $RECEIVE. The maximum value is 15 for other objects, except for the TMF transaction pseudofile (TFILE), which has a maximum of 1000. (For details about the TFILE, see the *TMF Application Programmer's Guide*.) If this parameter is omitted or 0, I/O operations are waited.

sync-or-receive-depth input

INT:value

The purpose of this parameter depends on the type of device being opened:

disk file	specifies the number of nonretryable (that is, write) requests whose completion the file system must remember. A value of 1 or greater must be specified to recover from a path failure occurring during a write operation. This value also implies the number of write operations that the primary process in a process pair can perform to this file without intervening checkpoints to its backup process. For disk files, this parameter is called sync depth. The maximum value is 15. If omitted, or if 0 is specified, internal checkpointing does not occur. Disk path failures are not automatically retried by the file system.
$RECEIVE file	specifies the maximum number of incoming messages read by READUPDATE[X] that the application process is allowed to queue before corresponding reply operations must be performed. If omitted or 0, READUPDATE[X] and reply operations to $RECEIVE are not permitted. For $RECEIVE, this parameter is called receive-depth, and the maximum number of queued incoming messages is 4047 in the H06.17/J06.06 and earlier RVUs. From H06.18/J06.07 RVU onwards, the maximum receive-depth value has been increased from 4047 to 16300.
process pair	specifies whether or not an I/O operation is automatically redirected to the backup process if the primary process or its processor module fails. For processes, this parameter is called sync depth. The maximum value is determined by the process. The value must be at least 1 for an I/O operation to a remote process pair to recover from a network failure.

If this parameter >= 1, the server is expected to save or be able to regenerate that number of replies.

If this parameter = 0, and if an I/O operation cannot be performed to the primary process of a process pair, an error indication is returned to the originator of the message. On a subsequent I/O operation, the file system redirects the request to the backup process.

For other device types, the meaning of this parameter depends on whether the sync-ID mechanism is supported by the device being opened. If the device does not support the sync-ID mechanism, 0 is used regardless of what you specify (this is the most common case). If the device supports the sync-ID mechanism, specifying a nonzero value causes the results of that number of operations to be saved; in case of path failures, the operations are retried automatically.

The actual value being used can be obtained by a call to FILE_GETINFOLIST_.

options input

INT:value

specifies optional characteristics. The bits, when set to 1, indicate:

<0>	Unstructured access. For disk files, access is to occur as if the file were unstructured, that is, without regard to record structures and partitioning. (For unstructured files, setting this bit to 1 causes secondary partitions to be inaccessible.) Must be 0 for other devices.
<1>	Nowait open processing. Specifies that the processing of the open proceed in a nowait manner. Unless FILE_OPEN_ returns an error, a nowait open must be completed by a call to AWAITIO[X]. This option cannot be specified for the Transaction Monitoring Facility (TMF) transaction pseudofile (TFILE). This option does not determine the nowait mode of I/O operations; that is controlled by the *nowait-depth* parameter. *nowait-depth* must have a nonzero value when this option is used.
<2>	No open time update. For disk files, the “time of last open” file attribute is not updated by this open. Must be 0 for other devices.
<3>	Any file number for backup open. When performing a backup open, specifies that the system can use any file number for the backup open. 0 specifies that the backup open is to have the same file number as the primary open. Error 12 is returned if that file number is already in use.
<4:9>	Reserved (specify 0)
<10>	Open an OSS file by its OSS pathname. Specifies that the file to be opened is identified by the *pathname* parameter.
<11>	Reserved (specify 0)

<12> No transactions. For $RECEIVE, messages are not to include transaction identifiers. Must be 0 if bit 15 is 1.

<13> I18N locale support. For $RECEIVE, data messages include internationalization locale information. Must be 0 if bit 15 is 1. For information about internationalization, see the *Software Internationalization Guide*.

<14> Old format system messages. For $RECEIVE, system messages should be delivered in C-series format. If this bit is 0, D-series format messages are delivered. For other device types, this bit must be 0. See Interprocess Communication Considerations on page 5-127

<15> No file-management system messages. For $RECEIVE, specifies that the caller does not wish to receive process open, process close, CONTROL, SETMODE, SETPARAM, RESETSYNC, and CONTROLBUF messages. If this bit is 0, messages are delivered as normal; some messages are received only with SETMODE 80. (Note that the meaning of this flag is opposite from that of the equivalent flag in the OPEN procedure). For other device types, this bit must be 0.

options (continued)

When *options* is omitted, 0 is used.

seq-block-buffer-id input

INT:value

if present and not 0, identifies the buffer to be used for shared sequential block buffering; all opens made through FILE_OPEN_ and using this ID share the same buffer. Any integer value can be supplied for this parameter.

If *seq-block-buffer-id* is omitted or 0, and sequential block buffering is requested, the buffer is not shared. In this case, the buffer resides in the process's process file segment (PFS) with the size given by *seq-block-buffer-len*.

seq-block-buffer-len input

INT:value

specifies whether sequential block buffering is being requested. If this parameter is supplied with a value greater than 0, it indicates a request for sequential block buffering and specifies the length in bytes of the sequential block buffer. If this parameter is omitted or 0, sequential block buffering is not requested. Sequential block buffering is only for disk files.

If this value is less than the data-block length that was given to this file or to any associated alternate-key file, the larger value is used. Supplying a nonzero value for this parameter causes a buffer to be allocated unless an existing buffer is to be shared (see the *seq-block-buffer-id* parameter). If an existing buffer is to be shared, but it is smaller than *seq-block-buffer-len*, sequential block buffering is not provided and a warning value of 5 is returned.

primary-processhandle input

INT .EXT:ref:10

indicates that the caller is requesting a backup open and specifies the process handle of the primary process that already has the file open when its backup attempts to open the file. If this parameter is supplied and not null (a null process handle has -1 in each word), *filenum* must contain the *filenum* value that was returned to the primary. If a null process handle is supplied, or the parameter is omitted, a normal open is being requested.

This option is used only when the backup process is the caller. It is more common for the primary to perform this operation by a call to FILE_OPEN_CHKPT_.

elections input

INT (32) :value, input

specifies these options:

<0:30> Reserved (specify 0).

<31> Use 64-bit primary keys. For disk files only, bit <31> specifies that 64-bit primary-key values are used instead of 32-bit values for unstructured, relative, or entry-sequenced files. Bit <31> is ignored for key-sequenced files and nondisk devices. The *elections* parameter can be used with both Enscribe format 1, Enscribe format 2, and OSS files.

If omitted, 0 is used.

General Considerations

- File numbers

  File numbers are unique within a process. The lowest file number is 0 and is reserved for $RECEIVE; the remaining file numbers start at 1. The lowest available file number is always assigned, except in the case of backup opens. When a file is closed, its file number becomes available for a subsequent file open to use.

- Maximum number of open files

  The maximum number of files in the system that can be open at any given time depends on the space available for control blocks: access control blocks (ACBs), file control blocks (FCBs), and open control blocks (OCBs). The amount of space available for control blocks is limited primarily by the physical memory size of the system. The maximum amount of space for ACBs is determined by the size of the process file segment (PFS). See the description of the *pfs-size* parameter under the PROCESS_CREATE_ Procedure (Superseded by PROCESS_LAUNCH_ Procedure).

- Multiple opens by the same process

  If a given file is opened more than once by the same process, a unique file number is returned for each open. These file numbers provide logically separate accesses to the same file; each file number has its own ACB, its own file position, and its own last error value. If a nowait IO operation is started and a second nowait operation is started (using a second file number for the same file), the IO requests are independent and may arrive in either order at the destination and may complete in either order.

Multiple opens on a given file can create a deadlock. This shows how a deadlock situation occurs:

```
error := FILE_OPEN_ ( myfile:len , filenuma ... );
! first open on file myfile.
     .
     .
error := FILE_OPEN_ ( myfile:len , filenumb ... );
! second open on file myfile.
     .
     .
error := FILE_OPEN_ ( myfile:len , filenumc ... );
! third open on file myfile.
     .
----
d       .
e  LOCKFILE ( filenumb, ... );     ! the file is locked
a       .                          ! using the file number
d       .                          ! associated with the
l       .                          ! second open.
o  READUPDATE ( filenumc, ... );   ! update the file
c       .                          ! associated with the
k       .                          ! third open.
----
```

Locks are granted on an open file (that is, file number) basis. Therefore, if a process has multiple opens of the same file, a lock of one file number excludes access to the file through other file numbers. The process is suspended forever if the default locking mode is in effect.

You now have a deadlock. The file number referenced in the LOCKFILE call differs from the file number in the READUPDATE call.

- Limit on number of concurrent opens

  There is a limit on the total number of concurrent opens permitted on a file. This determination includes opens by all processes. The specific limit for a file is dependent on the file's device type:

Disk Files	Cannot exceed 65,279 opens per disk
Process	Defined by process (see discussion of controlling openers in the *Guardian Programmer's Guide*)
$0	Unlimited opens
$0.#ZSPI	128 concurrent opens permitted
$OSP	10 times the number of subdevices (up to a maximum of 830 opens)
$RECEIVE	One open per process permitted
Other	Varies by subsystem

- Nowait I/O

  Specifying a *nowait-depth* value greater than 0 causes all I/O operations to be performed in a nowait manner. Nowait I/O operations must be completed by a call to AWAITIO[X]. Nowait IO operations on different file numbers (even if for the same file) are independent and may arrive in any order at the destination and may be completed by AWAITIO[X] in any order.

- Nowait opens

  If you open a file in a nowait manner (*options*.<1> = 1) and if FILE_OPEN_ returns no error (*error* = 0), the open operation must be completed by a call to AWAITIO[X]. If there is an error, no system message is sent to the object being opened and you do not need to call AWAITIO[X] to complete the operation.

  If there is no error, the *filenum* parameter returned by FILE_OPEN_ is valid. But you cannot initiate any I/O operation on the file until you complete the open by calling AWAITIO[X].

  If you specify the *tag* parameter in the call to AWAITIO[X], a -30D is returned; the values returned in the buffer and count parameters to AWAITIO[X] are undefined. If an error returns from AWAITIO[X], it is your responsibility to close the file.

  For the TMF transaction pseudofile, or for a waited file (*nowait-depth* = 0), a request for a nowait open is rejected.

  The file system implementation of a nowait open might use waited calls in some cases. However, it is guaranteed that the open message is sent nowait to a process; the opener does not wait for the process being opened to service the open message.

- Direct and buffered I/O transfers

  Except on NSAA systems, a file opened by FILE_OPEN_ uses direct I/O transfers, by default (user buffers).

  SETMODE 72 is used to override or explicitly set the buffer assignment for a file, that is, use either user buffers or process file segment (PFS) buffers for I/O transfers. This is unlike OPEN, which uses PFS buffers for I/O transfers, by default. For systems running H-Series RVUs, the default behavior is determined by the user_buffers flag in the object file, whether the USERIOBUFFER_ALLOW_ procedure is called, and whether this is an NSAA system.

  Calling the USERIOBUFFER_ALLOW_ procedure before the FILE_OPEN procedure will enable user buffers. The filesystem is still free to select the most efficient buffers to use. In practice, I/O less than 4096 bytes will use system (PFS) buffers.

  If system buffers are not used, you must ensure that you do not use or modify a buffer until a nowait I/O is completed. The only way to assure system buffers, is to use SETMODE 72,1 for that file.

  For more information, see Setmode 72 on page 14-80.

- Sequential block buffering

  Sequential block buffering is only supported for disk files. If sequential block buffering is used, the file should usually be opened with protected or exclusive access. Shared access can be used, but it is somewhat slower than the other access methods, and there might be concurrency problems. See the discussion of "Sequential Block Buffering" in the *Enscribe Programmer's Guide*.

- Named processes

  If you supply a process file name for a named process, it can represent any process with the same name. System messages are normally sent to the current primary process. The exception is when a named process supplies its own name to FILE_OPEN_. In that case the name refers to the backup process and system messages are sent there.

  A named process can be represented with or without a sequence number. FILE_OPEN_ treats the two name forms differently.

  - If you supply a process file name that includes a sequence number, the process must have a matching sequence number or the open fails with error 14. When retrying I/O on a process opened under such a name, the file system does not attempt to send messages to a possible backup process of the same name unless it has a matching sequence number. This is to assure that it is a true backup.
  - If you supply a process file name that does not include a sequence number, any process with a matching name can be opened and can be sent I/O retries. A newly created process that receives an I/O retry intended for another process of the same name will usually reject it with an error 60, but this is under the control of the application.

- Partitioned files

  A separate FCB exists for each partition of a partitioned file. There is one ACB per accessor (as for single-volume files), but this ACB requires more main memory since it contains the information necessary to access all of the partitions, including the location and partial-key value for each partition.

- Disk file open—security check

  When a disk file open is attempted, the system performs a security check. The accessor's (that is, the caller's) security level is checked against the file security level for the requested access mode, as follows:

for read access:	read security level is checked.
for write access:	write security level is checked.
for read-write access:	read and write security levels are checked.

  A file has one of seven levels of security for each access mode. (The owner of the file can set the security level for each access mode by using SETMODE function 1

or by using the File Utility Program SECURE command.) Table 5-5 shows the seven levels of security.

Table 5-5. Levels of Security

FUP Code	Program Values	Access
–	7	Local super ID only
U	6	Owner (local or remote), that is, any user with owner's ID
C	5	Member of owner's group (local or remote), that is, any member of owner's community
N	4	Any user (local or remote)
O	2	Owner only (local)
G	1	Member of owner's group (local)
A	0	Any user (local)

For a given access mode, the accessor's security level is checked against the file security level. File access is allowed or not allowed as shown in Table 5-6. In this table, file security levels are indicated by FUP security codes. For a given accessor security level, a Y indicates that access is allowed to a file with the security level shown; a hyphen indicates that access is not allowed.

Table 5-6. Allowed File Accesses

Accessor's Security Level	File Security Level						
	–	U	C	N	O	G	A
Super ID user, local access	Y	Y	Y	Y	Y	Y	Y
Super ID user, remote access	–	Y	Y	Y	–	–	–
Owner or owner's group manager, remote access	–	Y	Y	Y	–	–	–
Member of owner's group, remote access	–	–	Y	Y	–	–	–
Any other user, remote access	–	–	–	Y	–	–	–
Owner or owner's group manager, local access	–	Y	Y	Y	Y	Y	Y
Member of owner's group, local access	–	–	Y	Y	–	Y	Y
Any other user, local access	–	–	–	Y	–	–	Y

If the caller to FILE_OPEN_ fails the security check, the open fails with an error 48. A file's security can be obtained by a call to FILE_GETINFOLIST[BYNAME]_ , FILEINFO, or by the File Utility Program (FUP) INFO command.

If you are using the Safeguard product, this security information might not apply.

- Tape file open—access mode

The file system does not enforce read-only or write-only access for unlabeled tape, even though no error is returned if you specify one of these access modes when opening a tape file.

- File open—exclusion and access mode checking

  When a file open is attempted, the requested access and exclusion modes are compared with those of any opens already granted for the file. If the attempted open is in conflict with other opens, the open fails with error 12. Table 5-7 lists the possible current modes and requested modes, indicating whether an open succeeds or fails.

  For the Optical Storage Facility only, the “process exclusive” exclusion mode is also supported. Process exclusive is the same as exclusive for opens by other processes, but the same as shared for opens by the same process.

Note. Protected exclusion mode has meaning only for disk files. For other files, specifying protected exclusion mode is equivalent to specifying shared exclusion mode.

Table 5-7. Exclusion and Access Mode Checking

Exclusion Mode	Open attempted with: / Access Mode	File currently open with: Shared: Read/Write	Shared: Read Only	Shared: Write Only	Exclusive: Read/Write	Exclusive: Read Only	Exclusive: Write Only	Protected: Read/Write	Protected: Read Only	Protected: Write Only	File Closed
Shared	Read/Write	●	●	●							●
	Read Only	●	●	●				●	●	●	●
	Write Only	●	●	●							●
Exclusive	Read/Write	Always Fails									●
	Read Only										●
	Write Only										●
Protected	Read/Write		●								●
	Read Only		●						●		●
	Write Only		●								●

Legend

● = Open Successful

□ = Open Fails

Note: When a program file is running, it is opened with the equivalent of protected, read-only access.

CDT 008CDD

- Applications with large receive-depth values

  If you have applications that use large receive-depth values, you must periodically monitor their Message Quick Cell (MQC) usage levels using the `PEEK /CPU N/ MQCINFO` command in all processors to make sure that the total amount of memory allocated for MQCs does not approach the per-processor memory limit for MQCs. This limit is 128 MB in H06.19 / J06.08 and earlier RVUs, and 1 GB in H06.20/J06.09 and later RVUs. For more information, see Table J-3 on page 3.

  If you run applications with large receive-depth values on systems running H06.19/J06.08 or earlier RVUs, you must consider upgrading to H06.20/J06.09 or a later RVU if you notice MQC memory usage levels approach the per-processor

memory limit of 128 MB. To determine the amount of memory used for MQCs by CPU N from the `PEEK /CPU N/ MQCINFO` command output, add the page counts for all the MQC sizes, and then multiply the total page count allocated for MQCs by the page size (16 KB).

Disk File Considerations

- Maximum number of concurrent nowait operations

  The maximum number of concurrent nowait operations permitted for an open of a disk file is 1. Attempting to open a disk file and specify a *nowait-depth* value greater than 1 causes FILE_OPEN_ to fail with an error 28.

- Unstructured files

  - File pointers after an open

    After a disk file is opened, the current-record and next-record pointers begin at a relative byte address (RBA) of 0, and the first data transfer (unless positioning is performed) is from that location. After a successful open, the pointers are:

    current-record pointer = 0D
    next-record pointer = 0D

  - Sharing the same EOF pointer

    If a given disk file is opened more than once by the same process, separate current-record and next-record pointers are provided for each open, but all opens share the same EOF pointer.

- Structured files

  - Accessing structured files as unstructured files

    The unstructured access option (*options*.<0> = 1) permits a file to be accessed as an unstructured file. Note that the block format used by Enscribe must be maintained if the file is be accessed again in its structured form. (HP reserves the right to change this block format at any time.) For information about Enscribe block formats, see the *Enscribe Programmer's Guide*.

    For a file opened using the unstructured access option, a data transfer occurs to the position in the file specified by an RBA (instead of to the position indicated by a key address field or record number); the number of bytes transferred is that specified in the file-system procedure call (instead of the number of bytes indicated by the record format).

    If a partitioned file, either structured or unstructured, is opened using the unstructured access option, only the first partition is opened. The remaining partitions must be opened individually with separate calls to FILE_OPEN_ (each call specifying unstructured access).

    Accessing audited structured files as unstructured files is not allowed.

- Current-state indicators after an open

  After successful completion of an open, the current-state indicators have these values:

  - The current position is that of the first record in the file by primary key.
  - The positioning mode is approximate.
  - The comparison length is 0.

    If READ is called immediately after FILE_OPEN_ for a structured file, it reads the first record in the file; in a key-sequenced file, this is the first record by primary key. Subsequent reads, without intervening positioning, read the file sequentially (in a relative or entry-sequenced file) or by primary key (in a key-sequenced file) through the last record in the file.

    When a key-sequenced file is opened, KEYPOSITION is usually called before any subsequent I/O call (such as READ, READUPDATE, WRITE) to establish a position in the file.

    a position in the file.

- Queue files

  If the READUPDATELOCK operation is to be used, the value of the *sync-or-receive-depth* parameter must be 0. A separate open may be used for operations with *sync-or-receive-depth* > 0.

  Sequential block buffering cannot be used.

- 64-bit Primary keys

  In order to access non-key-sequenced files bigger than 4 GB, bit <31> of the FILE_OPEN_ *elections* parameter must be set. Use of this parameter allows the use of procedures using 32-bit primary keys (POSITION, KEYPOSITION, REPOSITION, GETSYNCINFO, and SETSYNCINFO) and the 32-bit key items of the FILE_GETINFOLIST_, FILEINFO, and FILERECINFO procedures.

Consideration for Terminals

The terminal being used as the operator console should not be opened with exclusive access. If it is, console messages are not logged.

Interprocess Communication Considerations

- Maximum concurrent nowait operations for an open of $RECEIVE

  The maximum number of concurrent nowait operations permitted for an open of $RECEIVE is 1. Attempting to open $RECEIVE and to specify a value greater than 1 causes an error 28 to be returned.

- When FILE_OPEN_ completes

When process A attempts to open process B, FILE_OPEN_ completes as follows:

- If process B has already opened $RECEIVE with file-management system messages disabled, the open call by process A completes immediately.
- If process B has opened $RECEIVE requesting file-management system messages enabled, the open call completes when process B reads the open message from process A by using READ[X], or if B uses READUPDATE[X], the open call completes when process B replies to the open message (by using REPLY[X]).

  If process B has not yet opened $RECEIVE, the open by process A does not complete until process B opens $RECEIVE. Specifically, the open by process A completes as follows:

  - When process B opens $RECEIVE with file-management system messages disabled, a waited open by process A completes immediately, but a nowait open by process A completes after the first read of $RECEIVE by process B.
  - When process B opens $RECEIVE with file-management system messages enabled, the open call by process A completes when process B reads the open message from A by using READ[X], or if B uses READUPDATE[X], the open call completes when process B replies to the open message (by using REPLY[X]).

- Message formats

  When $RECEIVE is opened by FILE_OPEN_, system messages are delivered to the caller in D-series format unless messages in C-series format are requested by setting *options*.<14> to 1. (No file-management system messages are delivered to the caller if *options*.<15> is set to 1 when opening $RECEIVE.)

- Messages from high-PIN processes

  Opening $RECEIVE with FILE_OPEN_ implies that the caller is capable of handling messages from processes with PINs greater than 255.

- Opening $RECEIVE and being opened by a remote long-named process

  If a process uses the FILE_OPEN_ procedure to open $RECEIVE and requests that C-series format messages be delivered (or if a process uses the OPEN procedure to open $RECEIVE), then a subsequent open of that process by another process on a remote node that has a process name consisting of more than five characters will fail with an error 20. Notification of this failure is not sent to the process reading $RECEIVE.

- Opening an unconverted (C-series format) process from a high-PIN process.

  A high-PIN process cannot open an unconverted process unless the unconverted process has the HIGHREQUESTERS object-file attribute set. If a high-PIN

process attempts to open a low-PIN process that does not have this attribute set, the high-PIN process receives file-system error 560.

System Message

When a process is opened by either FILE_OPEN_ or OPEN, it receives a process open message (unless it specified when opening $RECEIVE that it wants no messages). This message is in D-series format (message -103) or in C-series format (message -30), depending on what the receiving process specified when it opened $RECEIVE. The process handle of the opener can be obtained by a subsequent call to FILE_GETRECEIVEINFO_. For a description of the process open message, see the *Guardian Procedure Errors and Messages Manual*.

Note. This message is also received if the backup process of a process pair performs an open. Therefore, a process can expect two of these messages when being opened by a process pair.

DEFINE Considerations

- The *filename* or *pathname* parameter can be a DEFINE name; FILE_OPEN_ uses the file name given by the DEFINE as the name of the object to be opened. If you specify a CLASS TAPE DEFINE without the DEVICE attribute, the system selects the tape drive to be opened. A CLASS TAPE DEFINE has other effects when supplied to FILE_OPEN_ ; see Appendix E, DEFINEs for further information about DEFINEs.
- If a supplied DEFINE name is a valid name but no such DEFINE exists, the procedure returns an error 198 (missing DEFINE).
- When performing a backup open of a file originally opened through the use of a DEFINE, *filename* must contain the same DEFINE name. The DEFINE must exist and must have the same value as when the primary open was performed.

Safeguard Considerations

For information on files protected by Safeguard, see the *Safeguard Reference Manual*.

OSS Considerations

- To open an OSS file by its pathname, set *options*.<10> to 1 and specify the *pathname* parameter.
- OSS files can be opened only with shared exclusion mode.

Example

```
error := FILE_OPEN_ ( file^name:length, file^num );
```

The open in this example has these defaults; waited I/O, exclusion mode (shared), access mode (read/write), sync depth (0).

Related Programming Manuals

For programming information about the FILE_OPEN_ procedure, see the *Guardian Programmer's Guide* and the *Enscribe Programmer's Guide*.

FILE_OPEN_CHKPT_ Procedure

Summary

The FILE_OPEN_CHKPT_ procedure is called by a primary process to open a designated file for its backup process. These two conditions must be met before FILE_OPEN_CHKPT_ can be called successfully:

- The primary process must open the file.
- The backup process must be in the "monitor" state (that is, in a call to CHECKMONITOR).

The call to FILE_OPEN_CHKPT_ causes the CHECKMONITOR procedure in the backup process to call the FILE_OPEN_ procedure for the designated file.

Syntax for C Programmers

This passive backup procedure is not supported in C programs. For a comparison of active backup and passive backup, see the *Guardian Programmer's Guide*.

Syntax for TAL Programmers

```
error := FILE_OPEN_CHKPT_ ( filenum          ! i
                          ,[ status ] );   ! o
```

Parameters

error returned value

INT

is a file-system error number indicating the outcome of the checkpoint operation. Additional error information is returned in the *status* parameter.

filenum input

INT:value

is the number identifying the open file to be opened in the backup process. This value was returned by FILE_OPEN_ when the file was opened in the primary process.

status output

INT .EXT:ref

returns a value indicating the cause of the file-system error returned in *error*. Values are:

0 Backup open succeeded (*error* is 0)
1 File was opened in backup with warning
2 Open failed in backup
3 Unable to communicate with backup
4 Error occurred in primary

Considerations

- Identification of the backup process

  The system identifies the backup process to be affected by FILE_OPEN_CHKPT_ from the process's mom field in the process control block (PCB). For named process pairs, this field is automatically set up during the creation of the backup process.

- Nowait opens with FILE_OPEN_CHKPT_

  If a process is opened in a nowait manner (*options*.<1> = 1 in the call to FILE_OPEN_), the backup open is also performed in a nowait manner. It must be completed by a call to AWAITIO[X], in which case the *error* and *status* values are available through FILE_GETINFOLIST_ items 7 and 8, respectively. If you specify the *tag* parameter to AWAITIO[X], the returned value is -29D; the returned count and buffer address are undefined.

- Opens performed through the use of DEFINEs

  If the primary process opens a file through the use of a DEFINE, that DEFINE must exist unchanged when FILE_OPEN_CHKPT_ is called.

- Local setmode operations

  All local setmode operations (that is, setmodes that are supported by CHECKSETMODE) that have been applied to the file since the original open are also applied to the file by FILE_OPEN_CHKPT_.

- SQL/MX objects

  FILE_OPEN_CHKPT_ cannot be called for an SQL/MX object.

- See Considerations on page 5-111.

Example

```
error := FILE_OPEN_CHKPT_ ( file^number, status );
```

FILE_PURGE_ Procedure

Summary

The FILE_PURGE_ procedure deletes a disk file that is not open. When FILE_PURGE_ is executed, the disk file name is deleted from the volume's directory, and any disk space previously allocated to that file is made available to other files.

Syntax for C Programmers

```
#include <cextdecs(FILE_PURGE_)>

short FILE_PURGE_ ( const char *filename
                   ,short length );
```

Syntax for TAL Programmers

```
error := FILE_PURGE_ ( filename:length );    ! i:i
```

Parameters

error returned value

INT

is a file-system error number indicating the outcome of the operation.

filename:*length* input:input

STRING .EXT:ref:*, INT:value

specifies the name of the file to be purged. The value of *filename* must be exactly *length* bytes long and must be a valid disk file name or DEFINE name. If the name is partially qualified, it is resolved using the contents of the =_DEFAULTS DEFINE.

Considerations

- Purging a file audited by the Transaction Management Facility (TMF) subsystem

  If the file is audited by the TMF subsystem and if there are pending transaction-mode record locks or file locks, any attempt to purge the file fails with file-system error 12, whether or not openers of the file still exist.

  When an audited file is purged, all corresponding dump records are deleted from the TMF catalog. If the TMF subsystem is not active, attempts to purge an audited file fail with file-system error 82.

- Purging a partitioned file

  When you purge the primary partition of a partitioned file, the file system automatically purges all the other partitions located anywhere in the network that are marked as secondary partitions. A secondary partition is marked as such if it created at the same time as the primary partition.

- Security consideration

  When a file is purged, the data is not necessarily overwritten or erased, but pointers are changed to show the data to be absent. For security reasons, you might want to set the CLEARONPURGE flag for a file, using either function 1 of the SETMODE procedure or the File Utility Program (FUP) SECURE command. This flag causes all data to be physically erased (overwritten with zeros) when the file is purged.

- Expiration dates

  FILE_PURGE_ checks the expiration time of a file before purging it. If the expiration time is later than the current time, FILE_PURGE_ does not purge the file and returns file-system error 1091.

OSS Considerations

This procedure operates only on Guardian objects. If an OSS file is specified, error 1163 is returned.

SQL/MX Considerations

FILE_RENAME_ does not operate on SQL/MX objects. If a SQL/MX object is specified, file-system error 2 is returned.

Example

```
error := FILE_PURGE_ ( old^filename : old^filename^length );
```

Related Programming Manuals

For programming information about the FILE_PURGE_ procedure, see the *Guardian Programmer's Guide*.

FILE_RENAME_ Procedure

Summary

The FILE_RENAME_ procedure changes the name of an open disk file. If the file is temporary, assigning a name causes the file to be made permanent.

FILE_RENAME_ returns an error if there are incomplete nowait operations pending on the specified file.

Syntax for C Programmers

```
#include <cextdecs(FILE_RENAME_)>

short FILE_RENAME_ ( short filenum
                    ,const char *newname
                    ,short length );
```

Syntax for TAL Programmers

```
error := FILE_RENAME_ ( filenum                ! i
                       ,newname:length );      ! i:i
```

Parameters

error returned value

INT

is a file-system error number indicating the outcome of the operation.

filenum input

INT:value

is the number that identifies the open disk file to be renamed. The file number is obtained from FILE_OPEN_ or OPEN when the file is opened.

newname:*length* input:input

STRING .EXT:ref:*, INT:value

contains the file name to be assigned to the specified disk file. The value of *newname* must be exactly *length* bytes long. It must be a valid disk file name or the name of a DEFINE that designates a valid disk file name. If the file name is partially qualified, it is resolved using the contents of the =_DEFAULTS DEFINE.

Considerations

- Purge access for FILE_RENAME_

  The caller must have purge access to the file for the rename operation to be successful; otherwise, FILE_RENAME_ returns error 48 (security violation).

- Volume specification for *newname*

  The disk volume designated in *newname* (explicitly or implicitly) must be the same as the volume specified when opening the file. Neither the volume name nor the system name can be changed by FILE_RENAME_.

- System specification for *newname*

  If a system is specified as part of *newname*, it must be the same as the system name used when the file was opened.

- Partitioned files

  When the primary partition of a partitioned file is renamed, the file system automatically renames all other partitions located anywhere in the network.

- Renaming a file audited by the Transaction Management Facility (TMF) subsystem

  The file to be renamed cannot be a file audited by the TMF subsystem. An attempt to rename such a file fails with error 80 (invalid operation attempted on audited file or nonaudited disk volume).

- Structured files with alternate keys

  If the primary-key file is renamed, it remains linked with the alternate-key file. If you rename the alternate-key file and then try to access the primary-key file, an error 4 (failure to open an alternate key file) occurs because the primary-key file is still linked with the old name for the alternate-key file. You can use the File Utility Program (FUP) ALTER command to correct this problem.

- SQL/MX Objects

FILE_RENAME_ cannot be used with SQL/MX objects.

Safeguard Considerations

For information on files protected by Safeguard, see the *Safeguard Reference Manual.*

OSS Considerations

Error 564 (operation not supported on this file type) is returned if you attempt to rename an OSS file using the FILE_RENAME_ procedure.

Example

```
error := FILE_RENAME_ ( filenum, name:name^length );
```

Related Programming Manuals

For programming information about the FILE_RENAME_ procedure, see the *Guardian Programmer's Guide*.

FILE_RESTOREPOSITION_ Procedure

Summary

The FILE_RESTOREPOSITION_ procedure supersedes the REPOSITION Procedure (Superseded by FILE_RESTOREPOSITION_ Procedure) and is used to position a disk file to a saved position (the positioning information having been saved by a call to the FILE_SAVEPOSITION_ Procedure). The FILE_RESTOREPOSITION_ procedure passes the positioning block obtained by FILE_SAVEPOSITION_ back to the file system. Following a call to FILE_RESTOREPOSITION_, the disk file is positioned to the point where it was when FILE_SAVEPOSITION_ was called. Unlike the REPOSITION procedure, this procedure can be used with Enscribe format 2 files and OSS files greater than approximately 2 gigabytes as well as with other files.

A call to the FILE_RESTOREPOSITION_ procedure is rejected with an error if any incomplete nowait operations are pending on the specified file.

Syntax for C Programmers

```
#include <cextdecs(FILE_RESTOREPOSITION_)>

short FILE_RESTOREPOSITION_ ( short filenum
                            ,short * savearea
                            ,short savesize );
```

Syntax for TAL Programmers

```
error := FILE_RESTOREPOSITION_ ( filenum              ! i
                               ,savearea              ! i
                               ,savesize );           ! i
```

Parameters

error

INT:value

is a file-system error code that gives the status of the operation.

filenum input

INT:value

is the number that identifies the open disk file.

savearea input

INT:EXT.ref:*

is the positioning information from the FILE_SAVEPOSITION_ procedure. The size is given by the *savemax* parameter.

savesize input

INT:value

is the size in bytes of the information in the *savearea* parameter.

FILE_SAVEPOSITION_ Procedure

Summary

The FILE_SAVEPOSITION_ procedure supersedes the SAVEPOSITION Procedure (Superseded by FILE_SAVEPOSITION_ Procedure), and is used to save a disk file's current file positioning information in anticipation of a need to return to that position. The positioning information is returned to the file system in a call to the FILE_RESTOREPOSITION_ procedure when you want to return to the saved position. Unlike the SAVEPOSITION procedure, this procedure can be used with Enscribe format 2 files and OSS files greater than approximately 2 gigabytes as well as with other files.

Syntax for C Programmers

```
#include <cextdecs(FILE_SAVEPOSITION_)>

short FILE_SAVEPOSITION_ ( short filenum
                         ,short * savearea
                         ,short savemax
                         ,short * savesize );
```

Syntax for TAL Programmers

```
error := FILE_SAVEPOSITION_ ( filenum                    ! i
                            ,savearea                    ! o
                            ,savemax                     ! i
                            ,savesize);                  ! o
```

Parameters

error

INT:value

is a file-system error code that gives the status of the operation.

filenum input

INT:value

is the number that identifies the open disk file.

savearea output

INT:EXT.ref:*

is the location where the positioning information is received. The size is given by the *savemax* parameter.

savemax input

INT:value

specifies the *savemax* parameter size in bytes.

savesize output

INT:ref.EXT:1

returns the size in bytes of the information in the *savearea* parameter.

Considerations

- The size of the *savemax* parameter must meet these limits:
  - The size must be at least 44 bytes for non-key-sequenced files plus the sum of the primary-key length and 44 bytes for key-sequenced files.
  - For files with alternate keys, the size must be at least the primary-key length plus maximum alternate-key length and 44 bytes. The primary-key length is eight bytes for non-key-sequenced files and the maximum alternate-key length is the maximum value of all the alternate keys for the file.
  - For any currently supported file type, a *savemax* value of 300 is adequate.

FILE_SETKEY_ Procedure

Summary

The FILE_SETKEY_ procedure supersedes the KEYPOSITION[X] Procedures (Superseded by FILE_SETKEY_ Procedure). The FILE_SETKEY_ procedure is used to position by primary or alternate key within a structured file. However, positioning by primary key is usually done within key-sequenced files only when using this procedure; the FILE_SETPOSITION_ procedure is more commonly used for positioning by primary key within relative and entry-sequenced files. Unlike the KEYPOSITION[X] procedures, the FILE_SETKEY_ procedure expects the primary keys for relative and

entry-sequenced files to be 8 bytes long. Thus, this procedure can be used with format 2 files as well as with other files.

FILE_SETKEY_ sets the current position, access path, and positioning mode for the specified file. The current position, access path, and positioning mode define a subset of the file for subsequent access.

Syntax for C Programmers

```
#include <cextdecs(FILE_SETKEY_)>

short FILE_SETKEY_ (  short filenum
                    , char * key-value
                    , short key-len
                    , [ short keyspecifier ]
                    , [ short positioningmode ]
                    , [ short options ]
                    , [ short comparelength ] );
```

Syntax for TAL Programmers

```
error := FILE_SETKEY_ ( filenum                          ! i
                       ,key-value :key-value-len         ! i:i
                       ,[ keyspecifier ]                 ! i
                       ,[ positioningmode ]              ! i
                       ,[ options ]                      ! i
                       ,[ comparelength ] );             ! i
```

Parameters

error

INT:value

is a file-system error code that gives the status of the operation.

filenum input

INT:value

is the number that identifies an open-structured disk file.

key-value:key-value-len input, input

STRING.EXT:ref:*, INT:value

is the key value (with its length in bytes) to which the file is to be positioned.

keyspecifier input

INT:value

designates the key field (specified as the hexadecimal equivalent of the key identifier) to be used as the access path for the file:

keyspecifier	0, or if omitted, means use the file's primary key as the access path. Predefined key specifier for an alternate-key field means use that field as the access path.

positioningmode input

INT:value

indicates the type of key search to perform and the subset of records obtained.

These values are supported with the *positioningmode* parameter.

0 approximate

1 generic

2 exact

If *positioning-mode* is omitted, 0 is used. See the KEYPOSITION[X] Procedures (Superseded by FILE_SETKEY_ Procedure) for a detailed description of these values.

options input

INT:value

is a 16-bit value that specifies these options:

<0>	if 1, and if a record with exactly the *key-length* and *key-value* specified is found, the record is skipped. If the *keyspecifier* indicates a non-unique alternate key, the record is skipped only if both its alternate key and its primary key match the corresponding portions of the specified *key-value* (which should be an alternate key value concatenated with a primary key value) for *key-length* bytes (which should be the sum of the alternate and primary key lengths). This option is not supported for positioning by primary key in relative or entry-sequenced files.
<1>	return records in descending key order. (The file is read in reverse.)
<2>	specifies that positioning is performed to the last record in a set of records. This bit is ignored unless <1> is also set.

If the *options* parameter is omitted, 0 is used.

comparelength input

INT:value

is the length (in bytes) of the key used for comparing generic mode and exact-positioning mode that is used to decide when to stop returning these records. The value must be no longer than the *key-value-len* value. If omitted or 0, the value used is the smaller value of *key-value-len* or *keylength* of the key specified by the *keyspecifier* parameter.

Considerations

The considerations for the KEYPOSITION[X] Procedures (Superseded by FILE_SETKEY_ Procedure) apply to the FILE_SETKEY_ procedure.

FILE_SETLASTERROR_ Procedure

Summary

The FILE_SETLASTERROR_ procedure is used to set the error information for a file identified by the file number. This procedure can be used to set the Enscribe file-system error values: last-error, last-error detail, partition, and key in error.

Syntax for C Programmers

```
#include <cextdecs(FILE_SETLASTERROR_)>

short FILE_SETLASTERROR_ (  short filenum
                          , short errorcode
                          , [ short Errpart ]
                          , [ short Errkey ]
                          , [ short Errdetail ] );
```

Syntax for TAL Programmers

```
error := FILE_SETLASTERROR_ ( filenum                        ! i
                            ,errorcode              ! i
                            ,[ Errpart ]            ! i
                            ,[ Errkey ]             ! i
                            ,[ Errdetail ] );       ! i
```

Parameters

error returned value

INT:value

is a file-system error number that indicates the outcome of the operation.

filenum input

INT:value

is a number that identifies the open file of interest. *filenum* was returned by FILE_OPEN_ or OPEN when the file was originally opened.

You can also specify -1 for *filenum* to set the last-error value for a file that is not associated with a file number. See Considerations.

errorcode input

INT:value

is a value to set *last-error*.

Last-error indicates the file-system error number resulting from the last operation performed on the specified file. See Considerations.

errpart input

INT:value

is a number of the partition associated with the error for partitioned files.

errkey input

INT:value

is a key that specifies the key associated with the error for files with alternate keys.

errdetail input

INT:value

is a value to set the last-error detail that indicates the additional information, if available, for interpreting the errorcode. This value might be a file-system error number or another kind of value, depending on the operation and the primary error.

Considerations

- You can set the last-error, last-error detail, partition in error and key in error for a file that is not associated with a file number by specifying a filenum value of -1 to FILE_SETLASTERROR_.
- The parameters errpart, errkey and errdetail are set to a default value of 0, if the user does not pass these parameters.
- If the filenum is not corresponding to a valid open file, error 16 (FENOTOPEN) is returned.
- Error 590 (FEBADPARMVALUE) is returned if the user passes a value greater than 0 to any of these error values errorpart, errkey or errdetail when errorcode passed is 0 (FEOK).
- When FILE_SETLASTERROR_ is invoked, the file system remembers that the error values for that file have been overridden by this procedure.
- The values that were overridden can be obtained by calling FILE_GETINFOLIST_ with an item-code of info_ErrorExternallySet_. The file system remembers the values that were overridden when the FILE_SETLASTERROR_ was last called. The file-system error values for the file might have been set many times since then. For more information, see FILE_GETINFOLIST_ Procedure.

FILE_SETPOSITION_ Procedure

Summary

The FILE_SETPOSITION_ procedure supersedes the POSITION Procedure (Superseded by FILE_SETPOSITION_ Procedure). This procedure has the same function as the POSITION procedure but the FILE_SETPOSITION_ procedure

accepts an 8-byte record specifier. Thus, this procedure can be used with Enscribe format 2 files and OSS files greater than approximately 2 gigabytes as well as with other files.

The FILE_SETPOSITION_ procedure positions by primary key within relative and entry-sequenced files. For unstructured files, the FILE_SETPOSITION_ procedure specifies a new current position.

For relative and unstructured files, the FILE_SETPOSITION_ procedure sets the current position, access path, and positioning mode for the specified file. The current position, access path, and positioning mode define a subset of the file for subsequent access.

The FILE_SETPOSITION_ procedure is not used with key-sequenced files; the FILE_SETKEY_ procedure is used instead.

The caller is not suspended because of a call to the FILE_SETPOSITION_ procedure. A call to the FILE_SETPOSITION_ procedure is rejected with an error indication if there are incomplete nowait operations pending on the specified file.

Syntax for C Programmers

```
#include <cextdecs(FILE_SETPOSITION_)>

short FILE_SETPOSITION_ (  short filenum
                         , long long recordspecifier );
```

Syntax for TAL Programmers

```
error := FILE_SETPOSITION_ ( filenum                        ! i
                           ,recordspecifier );              ! i
```

Parameters

error

INT:value

is a file-system error code that gives the status of the operation.

filenum input

INT:value

is the number that identifies the opened file.

recordspecifier input

INT(64):value

is the 8-byte value that specifies the new setting for the current-record and next-record pointers.

Relative Files	The *recordspecifier* parameter is an 8-byte *record-num* value. A value of -2 specifies that the next write should occur at an unused record position. A value of -1 specifies that subsequent writes should be appended to the end-of-file location.
Unstructured Files	The *recordspecifier* parameter is an 8-byte *relative-byte-addr* value A value of -1 specifies that subsequent writes should be appended to the end-of-file location.

(For relative and unstructured files, the -1 and -2 remain in effect until a new *recordspecifier* is supplied.)

Entry-Sequenced Files	The *recordspecifier* parameter is an 8-byte record-addr (the primary key), whose format contains these elements: ● Block number (4 bytes) ● Relative record number within the block (4 bytes)

FILE_SETSYNCINFO_ Procedure

Summary

The FILE_SETSYNCINFO_ procedure supersedes the SETSYNCINFO Procedure (Superseded by FILE_SETSYNCINFO_ Procedure). Unlike the SETSYNCINFO procedure, this procedure can be used with Enscribe format 2 files and OSS files greater than approximately 2 gigabytes as well as with other files.

The FILE_SETSYNCINFO_ procedure is used by the backup process of a process pair after a failure of the primary process.

The FILE_SETSYNCINFO_ procedure passes a process pair's latest synchronization block (received in a checkpoint message from the primary) to the file system. Following a call to the FILE_SETSYNCINFO_ procedure, the backup process can retry the same series of write operations started by the primary process before its failure. The use of the sync block ensures that operations that might have been completed by the primary process before its failure are not duplicated by the backup.

Note. Typically, FILE_SETSYNCINFO_ is not called directly by application programs. Instead, it is called indirectly by CHECKMONITOR.

Syntax for C Programmers

```
#include <cextdecs(FILE_SETSYNCINFO_)>

short FILE_SETSYNCINFO_ (  short filenum
                         , short * infobuf
                         , short infosize );
```

Syntax for TAL Programmers

```
error := FILE_SETSYNCINFO_ ( filenum                    ! i
                           ,infobuf                     ! i
                           ,infosize );                 ! i
```

Parameters

error

INT:value

is a file-system error code that gives the status of the operation.

filenum input

INT:value

is the number that identifies the open file.

infobuf input

INT.EXT:ref:*

is the synchronization information returned by the FILE_GETSYNCINFO_ procedure.

infosize input

INT:value

is the size in bytes of the *infobuf* parameter.

FILE_WRITEREAD_ Procedure

Summary

The FILE_WRITEREAD_ procedure is similar to the WRITEREADX procedure except that it accepts two user buffers for the write and read operations respectively. This procedure supports the I/O operation only on the process type files.

The FILE_WRITEREAD_ procedure writes data to a process, which was previously opened, from an array in the application process, then waits for data to be transferred back from the process. The data buffers for the FILE_WRITEREAD_ procedure can either be in the caller's stack segment or in an extended data segment.

If the file is opened for nowait I/O, you must not modify the buffer before the I/O operation completes with a call to AWAITIOX. This condition also applies to other processes that may be sharing the segment. The application must ensure that the

readBuffer and writeBuffer used in the call to the FILE_WRITEREAD_ are not reused before the I/O operation completes with a call to AWAITIOX.

Note. The FILE_WRITEREAD_ procedure is supported only on systems running H-series RVUs, and is available only for native processes.

Interprocess Communication

The FILE_WRITEREAD_ procedure is used to originate a message to another process, which was previously opened, then waits for a reply from that process.

Syntax for C Programmers

```
#include <cextdecs(FILE_WRITEREAD_)>

short FILE_WRITEREAD_ (
     short filename,                    /* IN  */
     char _far * writeBuffer,           /* IN  */
     char _far * readBuffer,            /* OUT  */
     __int32_t writeCount,              /* IN  */
     __int32_t readCount,               /* IN  */
     __int32_t _far * countRead,        /* OUT OPTIONAL  */
     __int32_t   tag                    /* IN OPTIONAL  */
     );
```

Syntax for TAL Programmers

```
error := FILE_WRITEREAD_ ( INT filenum                  ! i
                         , STRING  .EXT writeBuffer     ! i
                         , STRING  .EXT readBuffer      ! o
                         , INT(32) writeCount           ! i
                         , INT(32) readCount            ! i
                         , INT(32) .EXT [ countRead ]   ! o
                           , INT(32) [ tag ] );         ! i
```

Parameters

error returned value

INT

is a file-system error number indicating the outcome of the operation.

filenum input

INT:value

is the number of an open file that identifies the file where the WRITE/READ is to occur.

writeBuffer input

STRING .EXT:ref:*

is an array containing information to be written to the file.

readBuffer output

STRING .EXT:ref:*

is an array containing information that was read from the file on return.

writeCount input

INT(32):value

is the number of bytes to be written:

{0:57344} for interprocess files

readCount input

INT(32):value

specifies the number of bytes to be read:

{0:57344} for interprocess files

countRead output

INT(32) .EXT:ref:1

is for waited I/O only. It returns a count of the number of bytes returned from the file into readBuffer.

tag input

INT(32):value

is for nowait I/O only. *tag* must uniquely identify the operation associated with this FILE_WRITEREAD_.

Note. The system stores the *tag* value until the I/O operation completes. The system then returns the *tag* information to the program in the *tag* parameter of the call to AWAITIO[X], thus indicating that the operation is complete.

Considerations

- Waited I/O READ

  If a waited I/O FILE_WRITEREAD_ is executed, the *countRead* parameter indicates the number of bytes actually read.

- Nowait I/O READ

  If a nowait I/O FILE_WRITEREAD_ is executed, the *countRead* has no meaning and can be omitted. The count of the number of bytes read is obtained when the I/O operation completes through the *count-transferred* parameter of the AWAITIOX procedure or the FILE_COMPLETE_ procedure.

  The FILE_WRITEREAD_ procedure must complete with a call to the AWAITIOX procedure or the FILE_COMPLETE_ procedure when used with a file that is opened nowait.

  You must not change the contents of the data buffers between the initiation and completion of a nowait FILE_WRITEREAD_ operation. This is because a retry can copy the data again from the writeBuffer and cause the wrong data to be written.

  You must avoid sharing a buffer between a FILE_WRITEREAD_ and another I/O operation because this creates the possibility of changing the contents of the data buffer before the write operation is completed.

- The *writeBuffer*, *readBuffer*, and *count-transferred* may be in the user stack or in an extended data segment. The buffers and count transferred cannot be in the user code space.
- If the *writeBuffer*, *readBuffer*, or *count-transferred* is in a selectable extended data segment, the segment must be in use at the time of the call. Flat segments allocated by a process are always accessible to the process.
- If the file is opened for nowait I/O, and buffer is in an extended data segment, you cannot deallocate or reduce the size of the extended data segment before the I/O operation completes with a call to the AWAITIOX procedure or to the FILE_COMPLETE_ procedure or is canceled by a call to CANCEL or CANCELREQ.
- If the file is opened for nowait I/O, the extended segment containing the buffers should be in use at the time of the call to the AWAITIOX procedure or the FILE_COMPLETE_ procedure.
- Nowait I/O initiated with these routines might be canceled with a call to CANCEL or CANCELREQ. The I/O operation is canceled if the file is closed before the I/O operation completes or if the AWAITIOX procedure or the FILE_COMPLETE_ procedure is called with a positive time limit and the specific file number and the request times out.
- If the extended address of the buffer is odd, bounds checking rounds the address to the next lower word boundary and checks an extra byte as well. The odd address is used for the transfer.

Errors for FILE_WRITEREAD

In addition to the other errors, the file-system error 22 is returned when:

- The address of a parameter refers to the selectable segment area but no selectable segment is in use at the time of the call.
- The address of a parameter references a privileged segment and the caller is not privileged.
- The file system cannot use the user's segment when needed.
- The writeBuffer and readBuffer memory are overlapping each other partially (however, if both the buffers point to the same address, it is acceptable).

Example

```
ERROR := FILE_WRITEREAD_( FILE^NUM, WRITEBUFFER, READBUFFER,
1, 72, NUM^READ );
```

The WRITEBUFFER contains the information to be written, and READBUFFER contains information that was read. In this case, 1 byte is to be written, and up to 72 bytes are to be read. The NUM^READ indicates how many bytes are read into the READBUFFER.

FILEERROR Procedure

Summary

The FILEERROR procedure is used to determine whether an I/O operation that completed with an error should be retried.

Syntax for C Programmers

```
#include <cextdecs(FILEERROR)>

short FILEERROR ( short filenum );
```

Syntax for TAL Programmers

```
status := FILEERROR ( filenum );              ! i
```

Parameters

status returned value

INT

indicates whether an I/O operation should be retried, as follows:

0 Operation should not be retried.
1 Operation should be retried.

filenum input

INT:value

is the number of an open file that identifies the file having the error.

Considerations

The FILEERROR procedure is called after a CCL return from a file-system procedure. The FILEERROR procedure determines if an operation should or should not be retried.

- If the error is caused by one of these:
  - A normal access request to a terminal currently in BREAK mode
  - BREAK key typed on a terminal where BREAK is enabled
  - Disk pack not up to speed

  FILEERROR delays the calling process for one second and then returns a 1, indicating a retry should be performed.
- If the error is an ownership error (error 200) or a path down error (error 201) and the alternate path is operable, FILEERROR returns a 1, indicating that the operation should be retried. If the alternate path is inoperable, a 0 is returned.
- If the error is caused by one of these:
  - A device not ready
  - No write-enable ring on a tape unit

- Paper out on a line printer

  An appropriate message is printed on the home terminal and is followed by a read operation from the terminal. If STOP is entered after the read (signaling that the condition cannot be corrected), FILEERROR returns a 0 to indicate that the operation should not be retried. If any other data is entered (typically, carriage return), it signals that the condition has been corrected, and FILEERROR returns a 1 to indicate that the operation should be retried.

- Any other error results in the file name, followed by the file-system error number, being printed on the home terminal. A 0 is returned, indicating that the operation should not be retried.

  If the file number has bit <0> set, no message is printed on the home terminal, unless *filenum* = -1.

  To prevent a message from being printed on the home terminal for *filenum* = -1, use *filenum* = %137777.

Example

```
error := 1;
WHILE error DO
  BEGIN
    CALL WRITE(fnum,buffer,count);
    IF < THEN
      BEGIN
        IF NOT FILEERROR(fnum) THEN CALL ABEND;
      END
    ELSE error := 0;
  END;
```

FILEINFO Procedure (Superseded by FILE_GETINFOLIST_ Procedure)

Note. The procedure that is superseded by FILE_GETINFO_ or FILE_GETINFOLIST_ does not support OSS ZYQ files.

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The FILEINFO procedure obtains error and characteristic information about a file. The file must be open if you refer to it by its file number, but if you refer to it by its file name, it need not be open. Table 5-8 on page 5-164 indicates which parameters are valid when specifying a file number or a file name.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
CALL FILEINFO ( [ filenum ]                              !  i
               ,[ error ]                                !  o
               ,[ file-name ]                            !  i,o
               ,[ ldevnum ]                              !  o
               ,[ devtype ]                              !  o
               ,[ extent-size ]                          !  o
               ,[ eof-pointer ]                          !  o
               ,[ next-record-pointer ]                  !  o
               ,[ last-modtime ]                         !  o
               ,[ filecode ]                             !  o
               ,[ secondary-extent-size ]                !  o
               ,[ current-record-pointer ]               !  o
               ,[ open-flags ]                           !  o
               ,[ subdevice ]                            !  o
               ,[ owner ]                                !  o
               ,[ security ]                             !  o
               ,[ num-extents-allocated ]                !  o
               ,[ max-file-size ]                        !  o
               ,[ partition-size ]                       !  o
               ,[ num-partitions ]                       !  o
               ,[ file-type ]                            !  o
               ,[ maximum-extents ]                      !  o
               ,[ unstructured-buffer-size ]             !  o
               ,[ more-flags ]                           !  o
               ,[ sync-depth ]                           !  o
               ,[ next-open-fnum ] );                    !  o
```

Parameters

filenum input

INT:value

is the number of an open file that identifies the file whose characteristics are to be returned. Either *filenum* or *file-name* must be specified; if both are passed, then *file-name* is set to the file name associated with *filenum*. If *filenum* is not specified, error 2 is returned for non-disk files.

error output

INT:ref:1

returns the error number associated with the last operation on the file. *filenum* or *file-name* can be specified with this parameter, but see the “Considerations” before passing these. For error recovery information, see the *Guardian Procedure Errors and Messages Manual*.

file-name input, output

INT:ref:12

specifies or returns the internal-format file name of this file. Either *filenum* or *file-name* must be passed. If both are passed, *file-name* returns with the name of the file associated with *filenum*; if *file-name* is passed without *filenum*, error 2 is returned for nondisk files.

ldevnum output

INT:ref:1 or INT:ref:16

is the logical device number of the device where this file resides. If the file is a process file, -1 is returned.

INT:ref:1 is used if the file is not partitioned or if *filenum* is omitted.

INT:ref:16 is used if the file is partitioned.

For partitioned files, an array of *ldevnum* is returned, one entry for each of 16 possible partitions:

[0] *ldevnum* of partition 0

[1] *ldevnum* of partition 1

.

.

.

[15] *ldevnum* of partition 15.

If -1 is returned for a partition, the partition is not open.

If the logical device number to be returned exceeds the maximum value allowed, the value 32767 is returned in *ldevnum* and 0 is returned in *error*, indicating that no error occurred. (The maximum value allowed for *ldevnum* is 65375, and should therefore be treated as unsigned. Note that no actual logical device will ever be assigned the value 32767.)

For partitioned files, *ldevnum* is set to 32767 and 0 is returned in *error* if at least one of the *ldevnum* values is greater than 65375.

devtype output

INT:ref:1

returns the device type and subtype of the device associated with this primary partition file. If *devtype*.<0> = 1, this device is a demountable disk volume. See Appendix A, Device Types and Subtypes for a list of device types and subtypes.

extent-size output

INT:ref:1

for disk files, returns the primary extent size in 2048-byte units. For nondisk devices, it returns the configured physical record length in bytes. For interprocess files, this parameter has no meaning. A returned value of -1 in this parameter means that the extent size cannot fit into this unsigned 2-byte parameter. The superseding procedure must be used to get the correct value.

eof-pointer output

INT(32):ref:1

for disk files only, returns the relative byte address (RBA) of the end-of-file location. A returned value of -1 in this parameter means that the end-of-file value cannot fit into this unsigned 4-byte parameter. The superseding procedure must be used to get the correct value.

next-record-pointer output

INT(32):ref:1

for disk files only, returns the next-record pointer setting:

relative files	A record number
entry-sequenced files	A record address
unstructured files	An RBA
key-sequenced files	Parameter is ignored (whatever is passed returns unchanged).

This parameter is not valid with *file-name*; use *filenum*. This parameter cannot be used with 64-bit primary keys. If an attempt is made, error 581 is returned.

last-modtime output

INT:ref:3

for disk files only, returns a three-word timestamp indicating the last time the file was modified; if the file has never been modified, its creation time is returned. *last-modtime* is of the same form as the *interval-clock* returned by TIMESTAMP and can be converted into a date by CONTIME.

You can obtain the same information in a four-word timestamp by calling FILEINQUIRE.

filecode output

INT:ref:1

for disk files only, returns the application-defined file code that is assigned when the file is created.

secondary-extent-size output

INT:ref:1

for disk files only, returns the size of the secondary file extents in 2048-byte units. A returned value of -1 in this parameter means that the extent size cannot fit into this unsigned 2-byte parameter. The superseding procedure must be used to get the correct value.

current-record-pointer output

INT(32):ref:1

for disk files only, returns the setting of the current-record pointer. This can be an even or odd value. This parameter is invalid with *file-name*; use *filenum*. This parameter cannot be used with 64-bit primary keys. If an attempt is made, error 581 is returned.

relative files	A record number
entry-sequenced files	A record address
unstructured files	An RBA
key-sequenced files	Parameter is ignored (whatever is passed returns unchanged).

open-flags output

INT:ref:1

returns the access granted when the file is opened. This parameter is invalid when used with *file-name*; use *filenum*. In this parameter:

<1> 1 For the $RECEIVE file only, means that the process wants to receive open, close, CONTROL, SETMODE, RESETSYNC, and CONTROLBUF messages.

<2> 1 Means unstructured access, regardless of the actual file structure (see OPEN Procedure (Superseded by FILE_OPEN_ Procedure)).

<3:5> Is the access mode:

0 read/write access
1 read-only access
2 write-only access

<6> 1 Indicates that resident buffers are provided by the application process for calls to file system I/O routines. A 0 is always returned in this bit (see OPEN Procedure (Superseded by FILE_OPEN_ Procedure)).

<8> 1 For process files means that the open message is to be sent nowait and must be completed by a call to AWAITIO.

<9> 1 Specifies that this is a queue file.

<10:11> Is the exclusion mode:

0 Shared access
1 Exclusive access
3 Protected access

<12:15> Is the maximum number of concurrent nowait I/O operations that can be in progress on this file at any given time. *open-flags*.<12:15> = 0 implies wait I/O.

subdevice output

INT:ref:1

returns the subdevice number associated with this file. Note that these values are only used internally by the operating system. This parameter is invalid with *file-name*; use *filenum*.

owner output

INT:ref:1

returns the identity of the file's owner in the form:

owner.<0:7> group number
owner.<8:15> member number

This parameter is invalid with *filenum*; use *file-name*.

security output

STRING:ref:5

returns the security assigned to the file. This parameter is invalid with *filenum*; use *file-name*. The fields have these meanings:

[0].<12> 1 Applies to a program file if the file has PROGID authority. When the program file is run, PROGID sets the caller's process access ID (PAID) to the owner's user ID of the program file.

[0].<13> 1 Applies if the CLEARONPURGE option is on for this file. If on, this option causes all data to be physically deleted from the disk when the file is purged. If this option is not on, the disk space is only logically deallocated when the file is purged, and no data is actually destroyed.

[0].<14> 1 Indicates there is a SAFEGUARD record for the file.

0 Indicates there is no SAFEGUARD record for the file. However, the file might still be protected by SAFEGUARD at the volume or subvolume level.

[1] Returns the reading security of the file.

[2] Returns the writing security of the file.

[3] Returns the execution security of the file.

[4] Returns the purging security of the file.

In *security*[1:4], the returned values are:

Returned Value	0	1	2	3	4	5	6	7
Security Level	A	G	O	n/a	N	C	U	Super

num-extents-allocated output

INT:ref:1

returns the number of extents that are allocated for the file. This parameter is invalid with *filenum*; use *file-name*.

max-file-size output

INT(32):ref:1

returns the maximum number of bytes configured for the file. This parameter is invalid with *filenum*; use *file-name*. A returned value of -1 in this parameter means that the true value cannot fit into this unsigned 4-byte parameter. The superseding procedure must be used to get the correct value.

partition-size output

INT:ref:1

returns the size in words of the area needed for the file's partition information array. This file partition information is retrieved from the *partition-parameters* array in the FILERECINFO procedure. This parameter is invalid with *filenum*; use *file-name*.

num-partitions output

INT:ref:1

returns the number of secondary partitions configured for the file. This parameter is invalid with *filenum*; use *file-name*.

file-type output

INT:ref:1

returns the file type and other information about the file. This parameter is invalid with *filenum*; use *file-name*. All bits are 0, except as described below:

<2> 1 For systems with the Transaction Management Facility, indicates this file is audited.

<5:7> Specifies object type for SQL object file:

0 File is not SQL
2 File is an SQL table
4 File is an SQL index

5 File is an SQL protection view
7 File is an SQL shorthand view

<9> 1 For interrogating queue files.

<10> 1 Indicates REFRESH is specified for this file.

<11> 1 For key-sequenced files, indicates index compression is specified.

<12> 1 For key-sequenced files, indicates data compression is specified.

1 For unstructured files, indicates ODDUNSTR is specified.

<13:15> Specifies the file structure:

0 Unstructured
1 Relative
2 Entry-sequenced
3 Key-sequenced

maximum-extents output

INT:ref:1

returns the maximum number of extents that can be allocated.

unstructured-buffer-size output

INT:ref:1

returns the internal buffer size to be used for an unstructured file.

more-flags output

INT:ref:1

returns various file attribute settings. Unless noted otherwise, these *more-flags* bits are valid with both the *file-name* and *filenum* parameters:

<0> 0 Verify writes off
1 Verify writes on (current file label default)

<1> 0 System automatically selects serial or parallel writes
1 Serial mirror writes only (current file label default)

<2> 0 Buffered writes enabled
1 write-thru (current file label default)

<3> 0 Audit-checkpoint compression off
1 Audit-checkpoint compression on (current file label default)

<4> 0 Crash-open flag off
1 Crash-open flag on (This is meaningful with the *file-name* parameter only.)

<5> 0 Rollforward needed flag off
1 Rollforward needed flag on

<6>	0	Broken file flag off
	1	Broken file flag on
<7>	0	File closed
	1	File is either open or has an incomplete TMF transaction against it. (This is meaningful with the *file-name* parameter only.)
<8>	0	Not licensed to have privileged procedures.
	1	Licensed to have privileged procedures.
<9>	0	Not a secondary partition of a partitioned file.
	1	Is a secondary partition of a partitioned file.
<10>	0	File contents are valid.
	1	File contents are probably invalid because a FUP DUP or LOAD, RESTORE, or similar operation ended abnormally. When this bit is 1, OPEN fails with error 59, although PURGE will work.
<11:15>		Reserved

sync-depth output

INT:ref:1

If this parameter is specified, the *filenum* parameter must be specified and must contain the number of an open file. FILEINFO returns the sync depth (or receive depth for $RECEIVE) of the file.

next-open-fnum output

INT:ref:1

If this parameter is specified, the *filenum* parameter must be specified and must contain the number of an open file or -1.

If an open file number is specified in *filenum*, FILEINFO returns the largest number of an open file whose file number is less than the file number specified in *filenum*. If there is no such file, FILEINFO returns -1.

If -1 is specified in *filenum*, FILEINFO returns the file number of the open file with the largest file number, or -1 if no files are currently open.

△ **Caution.** If the BUFFERED option is specified for an nonaudited file, a system failure or disk-process takeover (with *sync-depth* = 0) could cause the loss of buffered updates for the file that an application might not detect or handle properly unless modified.

Table 5-8 on page 5-164 indicates which FILEINFO parameters are valid when specifying the *filenum* or *file-name* parameter.

Table 5-8. FILEINFO filenum and file-name Parameters

Parameter	File Number	File Name
([< filenum >]	X	
, [< error >]	X	X
, [< filename >]	X	X
, [< ldevnum >]	X	X
, [< devtype >]	X	X
, [< extent-size >]	X	X
, [< eof-location >]	X	X
, [< next-record-pointer >]	X	
, [< last-modtime >]	X	X
, [< filecode >]	X	X
, [< secondary-extent-size >]	X	X
, [< current-record-pointer >]	X	X
, [< open-flags2 >]	X	
, [< subdev >]	X	
, [< owner >]		X
, [< security >]		X
, [< num-extents-allocated >]		X
, [< max-file-size >]		X
, [< partition-size >]		X
, [< num-partitions >]		X
, [< file-type >]		X
, [< maximum-extents >]	X	X
, [< unstructured-buffer-size >]	X	X
, [< more-flags >]	X	X
, [< sync-depth >]	X	
, [< next-open-fnum >])	X	

Condition Code Settings

< (CCL) indicates that an error occurred; the error number returns in *error*.

= (CCE) indicates that FILEINFO executed successfully.

> (CCG) indicates that an error occurred; the error number returns in *error*.

Considerations

- Specifying a file number or a file name

  If FILEINFO is called specifying *filenum*, the returned information is obtained from the access control block. If it is called specifying *file-name*, but not *filenum*, the returned information is obtained from the file label. There is no check to see if the file is actually opened when *file-name* is specified.

- Error returned when a file number is specified

  If FILEINFO is called specifying *filenum*, the last error associated with this file number is returned. If the file is opened more than once, only errors associated with *filenum* are returned; errors for the other opens are ignored. Note that the error returned might not originate from the last operation on the file. If there is an error in the call to FILEINFO, such as an incorrect parameter, that error is returned in *error*.

- Error returned when only a file name is specified

  If FILEINFO is called specifying *file-name*, but not *filenum*, only a small number of errors can be returned. No errors relating to any current open operation of the file can be returned. Typical errors are:

2	The file name is not a disk file name
11	File does not exist
13	Invalid file name
14	Device does not exist
249	Access to the system specified in *file-name* failed
250	The system specified in *file-name* is not part of the network

  In general, to obtain information about errors relating to operations on open files (including failures to open a file), use the file number form of this request. For information about files that do not have to be opened, use the file name form.

- Waited open that failed

  The error number of a preceding awaitio operation on any file or a waited open operation that failed can be obtained by passing a -1 in the *filenum* parameter. The error number returns in *error*.

- Disk file considerations

  The error number of a preceding CREATE or PURGE that failed can be obtained by passing a -1 in the *filenum* parameter. The error number returns in *error*.

- Calling FILEINFO subsequent to a CLOSE

  File-system error 16 (file not open) returns if FILEINFO is called subsequent to a CLOSE.

- FILEINFO and high-PIN processes

  If you use FILEINFO to request the file name of an unnamed high-PIN process that was opened using FILE_OPEN_, an error is returned.

- Calling FILEINFO after ENDTRANSACTION or ABORTTRANSACTION

  If there is a delay in the completion of a TMF transaction against a file after a call to ENDTRANSACTION or ABORTTRANSACTION, and if the file is closed by the last opener, there is a period of time during which FILEINFO reports that the file is open (*more-flags*.<7> = 1) and OPENINFO reports that the file is closed.

- Referencing Enscribe format 2 files with extent size greater than 65535 or OSS files larger than approximately 2 gigabytes

  If the file being referenced is an Enscribe format 2 file and the extent size exceeds 65535 or OSS file larger than approximately 2 gigabytes, item codes will return -1 with no error indication.

Examples

```
CALL FILEINFO ( FILENUM , ERROR );     ! get error of read
                                       ! operation.

CALL FILEINFO ( , , FILE^NAME , , , , , , , , , , , , OWNER );
```

FILEINQUIRE Procedure (Superseded by FILE_GETINFOLISTBYNAME_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The FILEINQUIRE procedure is used to obtain items of information about a file.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
CALL FILEINQUIRE ( [ filenumber ]                          ! i
                  ,[ file-name ]                           ! i
                  ,item-list                               ! i
                  ,number-items                            ! i
                  ,result-buffer                           ! o
                  ,result-buffer-length                    ! i
                  ,[ error-item ]                          ! o
                  ,[ error-code ] );                       ! o
```

Parameters

filenumber input

INT:value

identifies the file being inquired about. Required if *file-name* is not specified.

file-name input

INT:ref:12

is an internal-format file name that identifies the file being inquired about. Required if *filenumber* is not specified. May not be specified if *filenumber* is specified. A *define-name* can be given for this parameter.

item-list input

INT:ref:*

is an array in which the caller specifies the items of file information to be returned by the procedure. Each INT element of the array must contain a code from Table 5-9 on page 5-169.

number-items input

INT:value

is the number of items given in *item-list*.

result-buffer output

INT .EXT:ref:*

is the array in which the requested information items are returned. The items are returned in the order specified in *item-list*. Each item begins on an INT boundary; if the preceding item had a length of an odd number of bytes then an used byte will occur between the items. The length of each item is given in Table 5-9 on page 5-169.

result-buffer-length input

INT:value

is the size, in bytes, of the caller's *result-buffer*.

error-item output

INT:ref:1

if present, is returned the index of the item which was being processed when an error was detected. (The index of the first item in *item-list* is 0.)

error-code output

INT:ref:1

if present, is returned the status of the FILEINQUIRE call using standard file-system error codes.

Condition Code Settings

< (CCL) indicates that an error occurred (see *error-code*).

= (CCE) indicates that the FILEINQURE was successful.

> (CCG) indicates that one or more of the items requested are invalid for the file's device type, file type, open status, or other characteristic.

Considerations

- A particular item can be valid only when this procedure is called with *filenumber*, or only with *file-name*, or with either. This is indicated in Table 5-9.
- If error 2, operation not allowed for file type, occurs but no other errors occur (as indicated by condition code CCG), the other, valid items (if any) in the *item-list* still have their values returned. If an invalid item is of fixed size, the space for the item will be reserved in the *result buffer*, but the value is undefined. If an invalid item is of variable size, no space for it is reserved.
- If the *result buffer* is not large enough to hold the specified items, error 563 (buffer too small) will be returned.
- A call to FILEINQUIRE does not alter the saved error code associated with a file.
- For DP2 disk files, the values returned for time of file creation and last open are in the 4-word format for the time stamp. A peculiarity in the last open date could arise. The creation date for a file has been maintained since the first DP2 release, but the last open date and time has been maintained only since the B30 release of DP2. Therefore, it is possible that the date of last open for DP2 files could appear to be January 1, 4713 B.C. (returned as 0F). See Table 5-9.

Note. The information that items 6, 7, and 8 return is already available, either through FILEINFO or FILERECINFO, but some users may find the FILEINQUIRE interface more convenient.

Table 5-9. FILEINQUIRE Item Codes (page 1 of 3)

Code	Size (*bytes*)	Valid with Number/Name	Description
1	6	num	For labeled tapes, the tape volume serial number of the reel currently being processed
2	24	num	DEFINE name which was opened. Not valid for files which were opened without use of a DEFINE.
3	2	both	For key sequenced disk files, the number of levels used in the key indexing structure.

Table 5-9. FILEINQUIRE Item Codes (page 2 of 3)

Code	Size (*bytes*)	Valid with Number/Name	Description
4	2	both	For key sequenced disk files, the generic lock key length. For Enscribe files, if this value has never been set, the key length of the file is returned; for SQL tables, if this value has never been set, or if the set value is the same as the key length of the file, 0 is returned.
5	2	both	For structured disk files, the length in bytes of the *alternate-key-params* array.
6	*	both	For structured disk files, the *alternate- key-params* array. The length of this item is variable; its length can be determined through the use of item code 5.
7	2	both	For Enscribe disk files, the length in bytes of the *partition-params* array.
8	*	both	For Enscribe disk files, the *partition- params* array. The length of this item is variable; its length can be determined through the use of item code 7.
11	8	both	For DP2 disk files, the time (GMT) when the file was created.
12	8	both	For DP2 disk files, the time (GMT) of the most recent open of the file. If a file has never been opened, the time of its creation is returned. <15>: File is a program file containing compiled SQL statements. <14>: File is a program file containing compiled SQL statements and the compilation is assumed valid.
14	16	both	For disk files, the name of the SQL catalog subvolume associated with the file in internal form, or binary zeros if the file is not associated with an SQL catalog.

Table 5-9. FILEINQUIRE Item Codes (page 3 of 3)

Code	Size (*bytes*)	Valid with Number/Name	Description
15	8	both	Expiration date-time: For DP2 disk files, the time (GMT) beyond which purging of the file will be allowed. The value will be 0 if never set.
16	8	both	Last modification date-time: For DP2 disk files, the time (GMT) of the most recent modification to the file. If a file has never been modified, the time of its creation is returned. This is the same information as available through FILEINFO in 3-word TIMESTAMP form.
17	4	num	For particular access methods on some devices, a value associated with the last completed I/O operation. The meaning of the value is specific to the access method. For SNAX, it is the exception response identification number.

FILENAME_COMPARE_ Procedure

Summary

The FILENAME_COMPARE_ procedure compares two file names to determine whether they refer to the same object.

Syntax for C Programmers

```
#include <cextdecs(FILENAME_COMPARE_)>

short FILENAME_COMPARE_ ( const char *filename1
                         ,short length1
                         ,const char *filename2
                         ,short length2 );
```

Syntax for TAL Programmers

```
error := FILENAME_COMPARE_ ( filename1:length1     ! i:i
                            ,filename2:length2 );  ! i:i
```

Parameters

error returned value

INT

indicates the result of the operation. Possible values are:

-1 The file names do not refer to the same object.

0 The file names refer to the same object.

> 0 A file-system error prevented evaluation; the returned value is the file-system error number.

filename1:*length1* input:input

STRING .EXT:ref:*, INT:value

specifies the first file name that is compared. The value of *filename1* must be exactly *length1* bytes long. It must be a valid file name or valid DEFINE name; if it is a partially qualified file name, the contents of the =_DEFAULTS DEFINE are used to resolve it. See caution under “Considerations.”

filename2:*length2* input:input

STRING .EXT:ref:*, INT:value

specifies the second file name that is compared. The value of *filename2* must be exactly *length2* bytes long. It must be a valid file name or valid DEFINE

name; if it is a partially qualified file name, the contents of the =_DEFAULTS DEFINE are used to resolve it. See caution under “Considerations.”

Considerations

△ **Caution.** Passing an invalid file name to this procedure can result in a trap, a signal, or data corruption. To verify that a file name is valid, use the FILENAME_SCAN_ procedure.

- The name comparison is not case sensitive. For example, these file names refer to the same object:

```
\SYS99.$BIGVOL.MY.FILE
\sys99.$bigvol.my.file
```

- If one of the input parameters is a process name with the optional sequence-number field, and it is being compared to the same name without a sequence-number field, FILENAME_COMPARE_ considers the names equivalent provided that the named process currently exists and has the given sequence number; otherwise they are not considered equivalent. If both names have sequence-number fields, they must be the same for the file names to be considered the same.
- One or both of the file name parameters can be DEFINE names. For CLASS MAP DEFINEs, FILENAME_COMPARE_ uses the file name given by the DEFINE to make the comparison. For DEFINEs of other classes, a DEFINE name is considered equivalent only to the same DEFINE name.
- The FILENAME_COMPARE_ procedure compares whole file names. If you want to know whether two file names share a common part (for example, if they are on the same volume), one or more contiguous sections of each file name can be extracted by using the FILENAME_DECOMPOSE_ procedure before calling FILENAME_COMPARE_.

Example

```
status := FILENAME_COMPARE_ ( fname1:len1, fname2:len2 );
```

Related Programming Manual

For programming information about the FILENAME_COMPARE_ procedure, see the *Guardian Programmer’s Guide*.

FILENAME_DECOMPOSE_ Procedure

Summary

The FILENAME_DECOMPOSE_ procedure extracts and returns one or more parts of a file name or file-name pattern.

Syntax for C Programmers

```
#include <cextdecs(FILENAME_DECOMPOSE_)>

short FILENAME_DECOMPOSE_ ( const char *filename
                           ,short length
                           ,char *piece
                           ,short maxlen
                           ,short *piece-length
                           ,short level
                           ,[ short options ]
                           ,[ short subpart ] );
```

Syntax for TAL Programmers

```
error := FILENAME_DECOMPOSE_ ( filename:length     ! i:i
                              ,piece:maxlen        ! o:i
                              ,piece-length        ! o
                              ,level               ! i
                              ,[ options ]         ! i
                              ,[ subpart ] );      ! i
```

Parameters

error returned value

INT

returns a file-system error number indicating the outcome of the operation.

filename:*length* input:input

STRING .EXT:ref:*, INT:value

specifies the valid file name or file-name pattern from which a piece is extracted. The value of *filename* must be exactly *length* bytes long. See caution under "Considerations."

piece:*maxlen* output:input

STRING .EXT:ref:*, INT:value

returns the extracted piece of *filename*. *maxlen* is the length in bytes of the string variable *piece*.

piece-length output

INT .EXT:ref:1

returns the length in bytes of the extracted piece of *filename*. If an error occurs, 0 is returned.

level input

INT:value

specifies a part of *filename*. Together with *options* and *subpart*, it defines the piece of *filename* to be returned. Valid values are:

-1 Node name
0 Destination name (for example, volume, device, or process)
1 First qualifier (for example, subvolume)
2 Second qualifier (file identifier if disk file)

options input

INT:value

gives additional information about the piece of *filename* to be returned. The fields are:

<0:12> Reserved (specify 0).

<13> = 1 Do not return default values; that is, if a requested part is not present in *filename* but a default value exists for it, return a null string instead of the default value.

= 0 Default values can be returned.

<14> = 1 Include prefix, that is, the entire portion of *filename* that precedes the part specified by *level*.

= 0 Do not include prefix.

<15> = 1 Include suffix, that is, the entire portion of *filename* that follows the part specified by *level*.

= 0 Do not include suffix.

The default value is 0.

subpart input

INT:value

specifies a single section to be extracted from the level 0 (destination) part of *filename*. This parameter applies only to process file names, because only a process file name can have a level 0 part with multiple components. Valid values are:

0 Extract whole destination, that is, all sections occurring before the period (.).

1 Extract processor, that is, the numeric part designating the processor for an unnamed process.

2 Extract PIN, that is, the numeric part that gives the process identification number for an unnamed process.

3 Extract sequence number, that is, the numeric part that gives the sequence number of a process.

4 Extract name, that is, the alphanumeric section that begins with a dollar sign and ends at the first colon or period.

The default value is 0.

You should specify a nonzero value for *subpart* only when *level* is 0 and bits 14 and 15 (extract prefix and extract suffix) of *options* are 0.

Considerations

△ **Caution.** Passing an invalid file name or file-name pattern to this procedure can result in a trap, a signal, or data corruption. To verify that a file name or file-name pattern is valid, use the FILENAME_SCAN_ procedure.

- When the FILENAME_DECOMPOSE_ procedure returns a portion of a file name, it does not include leading or trailing separators (the characters . :). Internal separators between the parts of the returned portion are included. Special characters that are part of the name (the characters $ \ #) are always included.
- The *filename* parameter can contain a partially qualified file name or file-name pattern. Unless you specify *options*.<13> = 1 (no default values), the returned string reflects the resolution of the file name using the contents of your =_DEFAULTS DEFINE. If you request a name part that is absent after the file name has been resolved, either because the default values did not apply or because you specified *options*.<13> = 1, FILENAME_DECOMPOSE_ returns a value of 0 for *piece-length*.

Examples

This table shows some possible combinations of input values for calls to FILENAME_DECOMPOSE_ and the resulting output values. Note that “suffix” and “prefix” refer to the “include suffix” and “include prefix” options, respectively; “name” refers to a *subpart* value of 4. Assume that the current default values are “\SYS.$VOL.SUB”.

Input ***filename***	***level***	***options***	***subpart***	Output ***piece***
$a.b.c	0			$a
$a.b.c	0	suffix		$a.b.c
f	0			$VOL
f	0	suffix		$VOL.SUB.f
f	0	prefix		\SYS.$VOL
$p:4321.#a	0			$p:4321
$p:4321.#a	0		name	$p

Related Programming Manual

For programming information about the FILENAME_DECOMPOSE_ procedure, see the *Guardian Programmer’s Guide*.

FILENAME_EDIT_ Procedure

Summary

The FILENAME_EDIT_ procedure modifies one or more parts of a file name or file-name pattern, changing them to a specified value.

Syntax for C Programmers

```
#include <cextdecs(FILENAME_EDIT_)>

short FILENAME_EDIT_ ( char *filename
                      ,short maxlen
                      ,short *filename-length
                      ,const char *piece
                      ,short length
                      ,short level
                      ,[ short options ]
                      ,[ short subpart ] );
```

Syntax for TAL Programmers

```
error := FILENAME_EDIT_ ( filename:maxlen     ! i,o:i
                         ,filename-length     ! i,o
                         ,piece:length        ! i:i
                         ,level               ! i
                         ,[ options ]         ! i
                         ,[ subpart ] );      ! i
```

Parameters

error returned value

INT

returns a file-system error number indicating the outcome of the operation.

filename:*maxlen* input, output:input

STRING .EXT:ref:*, INT:value

on input, is the file name or file-name pattern to be edited; on output, contains the edited version of the file name or file-name pattern. The input must be a valid file name or valid file-name pattern; it must not be a DEFINE name. For the definitions of file name and file-name pattern. See caution under “Considerations. see Appendix D, File Names and Process Identifiers.

maxlen is the length in bytes of the string variable *filename*.

filename-length input, output

INT .EXT:ref:1

on input, is the length in bytes of the name to be edited; on output, is the length in bytes of the edited version of the file name.

piece:*length* input:input

STRING .EXT:ref:*, INT:value

specifies the string that is to replace the portion of *filename* that is indicated by the parameters that follow. If used, the value of *piece* must be exactly *length* bytes long. If *length* is 0, the indicated portion of *filename* is deleted. See "Considerations."

level input

INT:value

specifies a part of *filename*. Together with *options*, it defines the section of *filename* that is to be replaced. Valid values are:

-1 Node name
0 Destination name (for example, volume, device, or process)
1 First qualifier (for example, subvolume)
2 Second qualifier (file identifier if disk file)

options input

INT:value

gives additional options. If omitted, the default value is 0. The fields are:

<0:13> Reserved (specify 0).

<14> = 1 Remove *filename* prefix, that is, the entire portion of *filename* that precedes the part specified by *level*.

= 0 Do not remove prefix.

<15> = 1 Remove *filename* suffix, that is, the entire protion of *filename* that follows the part specified by *level*.

= 0 Do not remove suffix.

subpart input

INT:value

specifies a single section of the level 0 (destination) part of *filename* to be replaced. This parameter applies only to process file names, because only a process file name can have a level 0 part with multiple components. Valid values are:

0 Replace whole destination, that is, all sections occurring before the period (.).

1 Replace processor, that is, the numeric part designating the processor for an unnamed process.

2 Replace PIN, that is, the numeric part that gives the process identification number for an unnamed process.

3 Replace sequence number, that is, the numeric part that gives the sequence number of a process.

Note. The sequence number is mandatory for unnamed processes. Do not remove the sequence number from an unnamed process file name because a fatal error will result

.4 Replace name, that is, the alphanumeric section that begins with a question mark and ends at the first colon or period.

The default value is 0.

You should specify a nonzero value for *subpart* only when *level* is 0 and bits 14 and 15 (extract prefix and extract suffix) of *options* are 0.

Considerations

△ **Caution.** Passing an invalid file name or file-name pattern to this procedure can result in a trap, a signal, or data corruption. To verify that a file name or file-name pattern is valid, use the FILENAME_SCAN_ procedure.

- The value of *piece* that you supply to FILENAME_EDIT_ should include special characters that are part of the file name or file-name pattern (the characters $ \ # * ?) but not the leading or trailing separators (the characters . :). When you supply multiple parts in *piece*, you should include separators between the parts. These strings are examples of valid values for *piece*:

```
$S
\MYSYS
$DISK.*
$DISK.#TFILE
```

- You can request that a portion be deleted from *filename* by specifying that the length of *piece* be zero. When the portion is removed, any unneeded adjacent separators are also removed. A section can be removed from the middle of a file name, but the result (as with any modification) must be a valid file name or file-name pattern.

- When *filename* contains a partially qualified file name or file-name pattern, the contents of your =_DEFAULTS DEFINE are used to resolve it. This affects the operation of FILENAME_EDIT_ in two ways. First, it is possible to replace name parts that are not present in the input; one is replacing the implicit value with an explicit one. Second, an implicit name part might appear in the output, but this occurs only when it is necessary in order to form a valid file name. See these examples.

- You can use the *level* parameter to specify any part that is present in *filename* or implied through the =_DEFAULTS DEFINE. Generally, *level* cannot specify nonexistent right-hand (that is, higher) levels; the exception is that it can specify the level one greater than the highest level present in *filename*. Replacement of

this part effectively appends *piece* to *filename* using a period (.) as the separator.

- If the sequence number is removed from an unnamed file, an FE13 FEBADNAME error will result.

Examples

This table shows some possible combinations of input values for calls to FILENAME_EDIT_ and the resulting output values. Note that “suffix” refers to the “replace suffix” option in *options*, and “name” refers to a *subpart* value of 4. Assume that the current default values are “\SYS.$VOL.SUB”.

Input ***filename***	***piece***	***level***	**Modifiers**	Output ***filename***
$a.b.c	*	1		$a.*.c
$a.b.c	*	1	suffix	$a.*
$s	#mfile	1		$s.#mfile
f	x	1		x.f
f	$x	0		$x.SUB.f
f	\s	-1		\s.$VOL.SUB.f
$p:4321.#a	$z	0	name	$z:4321.#a

Related Programming Manual

For programming information about the FILENAME_EDIT_ procedure, see the *Guardian Programmer’s Guide*.

FILENAME_FINDFINISH_ Procedure

Summary

The FILENAME_FINDFINISH_ procedure releases the resources reserved for a search that was previously initiated by a call to FILENAME_FINDSTART_.

Syntax for C Programmers

```
#include <cextdecs(FILENAME_FINDFINISH_)>

short FILENAME_FINDFINISH_ ( short searchid );
```

Syntax for TAL Programmers

```
error := FILENAME_FINDFINISH_ ( searchid );  ! i
```

Parameters

error returned value

INT

is a file-system error number that indicates the outcome of the operation; error 16 indicates that *searchid* was not in use.

searchid input

INT:value

is the value that was previously returned by FILENAME_FINDSTART_ to identify the search request.

Related Programming Manual

For programming information about the FILENAME_FINDFINISH_ procedure, see the *Guardian Programmer's Guide*.

FILENAME_FINDNEXT_ Procedure

Summary

The FILENAME_FINDNEXT_ procedure returns the next name in a set of named entities that was defined by a call to the FILENAME_FINDSTART_ procedure.

Syntax for C Programmers

```
#include <cextdecs(FILENAME_FINDNEXT_)>

short FILENAME_FINDNEXT_ ( short searchid
                          ,[ char *name ]
                          ,[ short maxlen ]
                          ,[ short *name-length ]
                          ,[ short *entity-info ]
                          ,[ __int32_t tag ] );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.
- The parameter *maxlen* specifies the maximum length in bytes of the character string pointed to by *name*. The actual length of name in *name* is returned in *name-length*. These three parameters must either all be supplied or all be absent.

Syntax for TAL Programmers

```
error := FILENAME_FINDNEXT_ ( searchid              ! i
                             ,[ name:maxlen ]       ! o:i
                             ,[ name-length ]       ! o
                             ,[ entity-info ]       ! o
                             ,[ tag ] );            ! i
```

Parameters

error returned value

INT

is a file-system error number indicating the outcome of the operation. Error 1 (end of file) indicates that no more names are available.

searchid input

INT:value

is the value that was previously returned by FILENAME_FINDSTART_ to identify the search request.

name:*maxlen* output:input

STRING .EXT:ref:*, INT:value

contains the name being returned. The level of qualification of the name is determined by the *resolve-level* parameter previously supplied to FILENAME_FINDSTART_.

maxlen is the length in bytes of the string variable *name*.

name-length output

INT .EXT:ref:1

is the length in bytes of the value returned in *name*.

entity-info output

INT .EXT:ref:5

if present, returns a block of five words that might contain information about the entity designated by *name*. Note that some of the fields do not apply to all kinds of entities. (None of them apply to nodes.) The fields are:

[0] Device type.

[1] Device subtype.

[2:4] Device-specific information. When the device type is 3 (disk), the meanings are:

[2] Object type. If greater than 0, this is an SQL object; if equal to 0, this is a non-SQL disk file; if equal to -1, this is an entire volume or subvolume.

[3] File type. If the entity is a disk file, this is the file type (0 = unstructured, 1 = relative, 2 = entry-sequenced, 3 = key-sequenced); otherwise it is -1.

[4] File code. If the entity is a disk file, this is the file code given to the file; otherwise it is -1.

For other device types, words [2:4] are not currently defined.

tag input

INT(32):value

specifies a tag value that, when the procedure is used in a nowait manner, is returned in the completion message. If this parameter is omitted, the value 0D is used. If the procedure is not used in a nowait manner, this parameter is ignored. For details, see Nowait Considerations on page 5-185

Considerations

- The sequence of names returned by FILENAME_FINDNEXT_ does not have any specific order; in particular, the names might not come back in alphabetical order. Each kind of name part (for example, node or subvolume) has some order that is consistent for any RVU but might change from RVU to RVU.
- For a certain class of errors, you have the option of continuing the search even though some of the names cannot be returned. (This includes generic offline errors. For a discussion of generic offline errors, see FILENAME_FINDSTART_ Procedure.) Errors of this class can be recognized by the fact that even though an

error is indicated by the returned *error* value, a name is still returned in *name* and *name-length* is nonzero.

For these errors, the name returned is that of the entity associated with the error (for example, the node or device). It is generally not a name of the form being searched for but is a name to be used for error-reporting purposes. If you continue the search after one of these errors occurs by again calling FILENAME_FINDNEXT_, the set of names subordinate to the entity in error is skipped and the search proceeds to the next entity at that level. For example, if a device caused the error, a further search skips all subdevices on that device and proceeds to the next device.

In general, it is not worthwhile to retry errors that do not return a name, because the condition that caused the error is likely to recur.

For files residing on Storage Management Foundation (SMF) virtual disks, call FILE_GETINFOLISTBYNAME_ after the file name is returned by FILENAME_NEXT_. Without the additional call, FILENAME_FINDNEXT_ will always return 0 as object type.

Nowait Considerations

- If you specify the nowait option (*options*.<9> = 1) to FILENAME_FINDSTART_, the results are returned in the nowait FILENAME_FINDNEXT_ completion message (-109), not in the output parameters of the procedure. The format of this completion message is described in the *Guardian Procedure Errors and Messages Manual*. If *error* is not 0, no completion message is sent to $RECEIVE. Errors can be reported either on return from the procedure or through the completion message sent to $RECEIVE.
- Some errors are always returned in *error*. One of these is error 28, which occurs if you call FILENAME_FINDNEXT_ a second time on a particular *searchid* without having completed the previous call by reading the results from $RECEIVE.

Related Programming Manual

For programming information about the FILENAME_FINDNEXT_ procedure, see the *Guardian Programming Reference Summary*.

FILENAME_FINDSTART_ Procedure

Summary

The FILENAME_FINDSTART_ procedure sets up a search of named entities. After specifying search criteria to FILENAME_FINDSTART_, you call the FILENAME_FINDNEXT_ procedure to retrieve individual names.

Syntax for C Programmers

```
#include <cextdecs(FILENAME_FINDSTART_)>

short FILENAME_FINDSTART_ ( short *searchid
                          ,[ const char *search-pattern ]
                          ,[ short length ]
                          ,[ short resolve-level ]
                          ,[ short device-type ]
                          ,[ short device-subtype ]
                          ,[ short options ]
                          ,[ const char *startname ]
                          ,[ short length ] );
```

- The character-string parameters *search-pattern* and *startname* are each followed by a parameter *length* that specifies the length in bytes of the character string. In each case, the character-string parameter and the corresponding length parameter must either both be supplied or both be absent.

Syntax for TAL Programmers

```
error := FILENAME_FINDSTART_ ( searchid                    ! o
                             ,[ search-pattern:length ]    !
i:i
                             ,[ resolve-level ]            ! i
                             ,[ device-type ]              ! i
                             ,[ device-subtype ]           ! i
                             ,[ options ]                  ! i
                             ,[ startname:length ] );      !
i:i
```

Parameters

error returned value

INT

returns a file-system error number indicating the outcome of the operation.

searchid output

INT .EXT:ref:1

returns a value that identifies the search request for other file-name inquiry procedures.

search-pattern:*length* input:input

STRING .EXT:ref:*, INT:value

if supplied and if *length* is not 0, contains a valid file-name pattern that specifies the set of names to be searched. If used, the value of *search-pattern* must be exactly *length* bytes long. A partially qualified file-name pattern is resolved using the contents of the caller's =_DEFAULTS DEFINE. "*" is the default search pattern.

For the definition of file-name pattern, see Appendix D, File Names and Process Identifiers.

resolve-level input

INT:value

if present, indicates how fully qualified the names returned by FILENAME_FINDNEXT_ are to be. The value indicates the part that should be the leftmost component of a returned name. If this parameter is omitted, the default value is -1. Valid values are:

-1 Node name
0 Destination name (for example, volume, device, or process)
1 First qualifier (for example, subvolume)
2 Second qualifier (file identifier if disk file)

The specified level must not be further to the right than the level in *search-pattern* that contains the first asterisk (*) or question mark (?).

device-type input

INT:value

if present and not equal to -1, specifies a device-type value to be used in selecting returned names. A name returned by FILENAME_FINDNEXT_ must represent, or must not represent (depending on the value of *options*), an entity of the specified device type (or an entity that has a parent device of that device type). If this parameter is omitted or equal to -1, device type is disregarded when selecting file names. See Device Type Considerations on page 5-190.

device-subtype input

INT:value

if present and not equal to -1, specifies a device-subtype value to be used in selecting returned names. A name returned by FILENAME_FINDNEXT_ must

represent, or must not represent (depending on the value of *options*), an entity of the specified device subtype. If this parameter is omitted or equal to -1, device subtype is disregarded when selecting file names. See Device Type Considerations on page 5-190.

options input

INT:value

specifies further options. The bits, when equal to 1, indicate:

Bit	Meaning
<0:7>	Reserved (specify 0)
<8>	If a physical file corresponding to a NonStop Storage Management Foundation (SMF) logical file is encountered, the name of the physical file is to be returned.
<9>	The search is to be executed in a nowait manner. When this option is specified, some parts of the search processing are asynchronous with respect to the caller, and the results of the search are returned in system messages sent to $RECEIVE rather than through the output parameters of FILENAME_FINDNEXT_. See “Considerations.”
<10>	Device simulation by subtype 30 processes is not to be supported.
<11>	The search is not to include subprocesses; that is, the search is to ignore qualifier names that are subordinate to names of user processes (other than device simulation processes). Because there is no search for qualifier names, the subordinate name inquiry system message (-107) is not sent by FILENAME_FINDNEXT_. The PROCESS_SETINFO_ procedure enables the receipt of this system message. For more information on this system message, see the *Guardian Procedure Errors and Messages Manual*.
<12>	If an entity is encountered that is offline (that is, the system is not connected or the device is down), an error is to be reported. When this bit is zero, such entities are sometimes ignored. See Error Handling on page 5-191.
<13>	If *device-subtype* is supplied, a file name must *not* match the device subtype value in order to be returned.
<14>	If *device-type* is supplied, a file name must *not* match the device type value in order to be returned.
<15>	If *startname* is supplied, and if the first name returned would be *startname*, then that name is to be skipped and this name should be returned.

The default value is 0. See “Considerations” and Device Type Considerations on page 5-190.

startname:*length* input:input

STRING .EXT:ref:*, INT:value

if supplied and if *length* is not 0, specifies a file name that indicates where, within the set of file names that meet the search criteria, selection should begin. If used, the value of *startname* must be exactly *length* bytes long. It must be a valid file name and must fall within the set of names described by *search-pattern*. Unless *options*.<15> = 1, the first call to FILENAME_FINDNEXT_ will return this name (if it exists).

Considerations

- You must call FILENAME_FINDSTART_ to initiate a search for named entities. If no error occurs, FILENAME_FINDSTART_ returns a *searchid* value that you use to identify the particular search when making subsequent calls to FILENAME_FINDNEXT_. A process can have up to 16 concurrent searches. (Having more than 16 searches causes FILENAME_FINDSTART_ to fail with error 34.) When finished searching, you should call FILENAME_FINDFINISH_ with *searchid* to release the resources.
- The file-name pattern supplied in *search-pattern* determines the kind of names that will be returned by FILENAME_FINDNEXT_ and also restricts the range of name values. For example, * will return node names; $* will return device names and process file names. Subvolume names can be retrieved with file-name patterns such as $VOL.*.

  More than one level in the file-name pattern can contain asterisks and question marks. Note that a file-name pattern such as *.*.* designates not only disk files but also I/O subdevices and processes that have two levels of qualifiers.

  For the definition of file-name pattern, see Appendix D, File Names and Process Identifiers.
- A search for qualifier names of a process (for example, qualifier #TERM1 of the process $TERM.#TERM1) can be performed if both of these are true:
  - The process that is the target of the search called the PROCESS_SETINFO_ procedure and set attribute 49 to 1 to enable the receipt of the subordinate name inquiry system message (-107).
  - *options*.<11> of the FILENAME_FINDSTART_ procedure is set to 0.

  For descriptions of the messages and replies that must be supported to search for qualifier names, see the *Guardian Procedure Errors and Messages Manual*.
- The names returned by FILENAME_FINDNEXT_ are returned in a sequence that is not necessarily alphabetic. (See Considerations on page 5-184.) You can specify a starting point in this sequence other than the normal one by using the *startname* parameter of FILENAME_FINDSTART_ and thereby avoid

processing some initial portion of the sequence. You can do this, for instance, to restart a discontinued search from the point where it stopped.

- When using the nowait option (*options*.<9> = 1), you still call FILENAME_FINDNEXT_ for each name but the results are returned in a system message to $RECEIVE rather than in the output parameters of FILENAME_FINDNEXT_. For the format of this system message, see the *Guardian Procedure Errors and Messages Manual*.

  The nowait interface guarantees only that device-type simulation and subname inquiries to user processes are asynchronous to the caller; any other part of a search might be synchronous (that is, might execute while the caller waits) or asynchronous in a given software release.

- The FILENAME_FIND* procedures can be used to search for files on SMF virtual volumes. However, when searching disk volumes, the names in the special SMF subvolumes (ZYS* and ZYT*) where SMF physical files reside are not returned by the FILENAME_FINDNEXT_ procedure except when the search pattern supplied to the FILENAME_FINDSTART_ procedure includes “ZYS” or “ZYT” as the first three characters of the subvolume portion of the pattern, or when *options*.<8> is set equal to 1.

Device Type Considerations

- The *device-type* parameter can be used to restrict the set of names that are returned. If it is supplied, a name must represent an entity of the specified device type to be returned. If *options*.<14> is equal to 1, the meaning of *device-type* is reversed: all names are returned except those representing entities of the specified device type. The *device-subtype* parameter acts in the same manner with respect to the device subtype. These parameters do not apply to system name searches. A typical use might be to restrict a file-name pattern such as *.*.* to disk files by supplying a *device-type* value of 3.

- Note that if the *device-type* value is 3, the subordinate name inquiry system message (-107) is never sent, regardless of the setting of *options*.<11>.

- The system allows certain processes, which are distinguished by having a device subtype of 30, to simulate device types. During a file name search, these processes are normally sent a system message inquiring about the device-type and subtype values they present. The result of this inquiry is used for selection under the *device-type* and *device-subtype* selection criteria and for information reporting by FILENAME_FINDNEXT_.

  You can suppress device type simulation by specifying *options*.<10> = 1. Without device-type simulation, all simulator processes and their subprocesses show a device type of 0 and a subtype of 30.

- When searching for only disk files, the search is usually more efficient if you use the *device-type* parameter to restrict the search to disk devices. Otherwise, time is spent making inquiries to nondisk devices (which can have subdevices of

various device types that must also be searched) and simulator processes (which can have subprocesses of various simulated device types that must also be searched).

Error Handling

A section of a search pattern can be termed **generic** if it does not designate a specific entity (that is, if the section contains an asterisk or a question mark, or if the section is to the right of such a character). For example, the destination section $* is generic; the destination section in \sys.$dev.* is not generic. In *.$dev, both the node and the destination sections are generic.

Some entities that should be inspected during a search might be offline. This can occur because either a node is not connected (error 250) or a device is down (errors 62-66). But if the section of the search pattern corresponding to the offline entity is generic, the normal action of FILENAME_FINDNEXT_ is to bypass the offline entity without reporting the error. Such an error is termed a generic offline error. You can cause FILENAME_FINDNEXT_ to report all offline errors, including generic offline errors, by specifying *options*.<12> = 1. (Note that it is possible that some remote nodes are not known to the local node because the local node has not communicated with them since the node was system loaded; offline errors are not reported in such cases, regardless of the value of *options*.)

Errors associated with entities that are designated explicitly (that is, not generically) are always reported. This includes not only offline errors but also such errors as error 18 (node does not exist) and error 14 (device does not exist).

Example

```
!  process all 6-byte file names in the current subvolume
error := FILENAME_FINDSTART_ ( searchid,, 2 ); ! return only
                                               ! level 2 name
                                               ! part
IF NOT error THEN
   error := FILENAME_FINDNEXT_ ( searchid, name:128,
            namelen );
WHILE NOT error DO
   BEGIN
   IF namelen = 6 THEN ...  ! process name !
   error := FILENAME_FINDNEXT ( searchid, name:128,
            namelen );
   END;
error := FILENAME_FINDFINISH_ ( searchid );
```

Related Programming Manual

For programming information about the FILENAME_FINDSTART_ procedure, see the *Guardian Programmer's Guide*.

FILENAME_MATCH_ Procedure

Summary

The FILENAME_MATCH_ procedure determines whether one or more contiguous sections of a file name match the corresponding sections of a file-name pattern (that is, whether the file name sections might be represented by the pattern sections). FILENAME_MATCH_ does not resolve partially qualified file names.

Syntax for C Programmers

```
#include <cextdecs(FILENAME_MATCH_)>

short FILENAME_MATCH_ ( const char *filename
                       ,short length
                       ,const char *pattern
                       ,short length
                       ,[ short *generic-set ] );
```

Syntax for TAL Programmers

```
status := FILENAME_MATCH_ ( filename:length     ! i:i
                           ,pattern:length      ! i:i
                           ,[ generic-set ] );  ! o
```

Parameters

status returned value

INT

returns a value indicating the result of the operation. Valid values are:

2 Match; the name fits the pattern.

1 Partial match; the name fits the left-hand pattern sections.

0 No match; the name does not fit the pattern.

<0 An error occurred; one of these values is returned:

-1 Missing *filename* parameter

-2 Missing *pattern* parameter
-3 Length error on *filename* parameter
-4 Length error on *pattern* parameter
-5 Bounds error on *generic-set* parameter

filename:*length* input:input

STRING .EXT:ref:*, INT:value

is one or more contiguous sections of a valid file name that is to be tested for a match with *pattern*. The value of *filename* must be exactly *length* bytes long. It must have the same level of left-hand qualification as *pattern* or else a match cannot result (for example, if *pattern* starts with a node-name section, *filename* must have a node name). See caution under “Considerations.”

pattern:*length* input:input

STRING .EXT:ref:*, INT:value

is one or more contiguous sections of a valid file-name pattern to be matched. The value of *pattern* must be exactly *length* bytes long. For the definition of file-name pattern, see Appendix D, File Names and Process Identifiers. See caution under “Considerations.”

generic-set output

INT .EXT:ref:1

if *status* is 0, returns a value indicating whether *filename* is a member of, falls before, or falls after the generic set of names defined by *pattern*. A name is part of the generic set if it matches *pattern* up to the first wild-card character (* or ?). Valid values are:

-1 The name falls before the first possible match.
0 The name falls within the set of possible matches.
1 The name falls after the last possible match.

Considerations

△ **Caution.** Passing an invalid file name or file-name pattern to this procedure can result in a trap, a signal, or data corruption. To verify that a file name or file-name pattern is valid, use the FILENAME_SCAN_ procedure.

- *filename* and *pattern* must have the same level of left-hand qualification. For example, if *pattern* has a node-name section, *filename* matches only if it contains a node name. FILENAME_MATCH_ does not support file-name resolution with current default values; you can use the FILENAME_RESOLVE_ procedure to resolve partially qualified file names before calling FILENAME_MATCH_.
- Matching is syntactic only and does not involve lookups of actual processes, logical devices, or other entities.

- FILENAME_MATCH_ does not use the process or system context (as defaulting would require), so it can be used in environments where the PFS (process file segment) is not available.

Examples

```
result := FILENAME_MATCH_ ( filename:flen, pattern:plen );
```

This table shows some possible combinations of input parameters and the corresponding values of *status* for the foregoing call to FILENAME_MATCH_:

filename	***pattern***	***status***	***generic-set***
cab.ride	C*B.*	2 (match)	N/A
cab	c*b.*	1 (partial match)	N/A
bable	c*b.*	0 (no match)	-1 (before set)
c	c*b.*	0 (no match)	0 (part of set)
czz.ride	c*b.*	0 (no match)	0 (part of set)
d	c*b.*	0 (no match)	+1 (after set)

Related Programming Manual

For programming information about the FILENAME_MATCH_ procedure, see the *Guardian Programmer's Guide*.

FILENAME_RESOLVE_ Procedure

Summary

The FILENAME_RESOLVE_ procedure converts a partially qualified file name to a fully qualified file name. You can supply a search list when qualifying a disk file name. You can also use FILENAME_RESOLVE_ to resolve absent sections of a file-name pattern or to resolve a DEFINE that contains a file name and can be opened. For the definitions of file name and file-name pattern, see Appendix D, File Names and Process Identifiers. For further information about DEFINEs, see Appendix E, DEFINEs.

Syntax for C Programmers

```
#include <cextdecs(FILENAME_RESOLVE_)>

short FILENAME_RESOLVE_ ( const char *partialname
                         ,short length
                         ,char *fullname
                         ,short maxlen
                         ,short *fullname-length
                         ,[ short options ]
                         ,[ const char *override-name ]
                         ,[ short length ]
                         ,[ const char *search ]
                         ,[ short length ]
                         ,[ const char *defaults ]
                         ,[ short length ] );
```

- Some character-string parameters to FILENAME_RESOLVE_ are followed by a parameter *length* that specifies the length in bytes of the character string (not counting the null-byte terminator). Where the parameters are optional, the character-string parameter and the corresponding length parameter must either both be supplied or both be absent.

Syntax for TAL Programmers

```
error := FILENAME_RESOLVE_ ( partialname:length          !
i:i
                            ,fullname:maxlen             !
o:i
                            ,fullname-length             ! o
                            ,[ options ]                 ! i
                            ,[ override-name:length ]    !
i:i
                            ,[ search:length ]           !
i:i
                            ,[ defaults:length ] );      !
i:i
```

Parameters

error returned value

INT

is a file-system error number indicating the outcome of the operation.

partialname:*length* input:input

STRING .EXT:ref:*, INT:value

is a valid, partially qualified file name or file-name pattern that is to be resolved. *partialname* can also be a valid DEFINE name (see the description of the *options* parameter). The value of *partialname* must be exactly *length* bytes long. See caution under “Considerations.”

fullname:*maxlen* output:input

STRING .EXT:ref:*, INT:value

contains the resulting fully qualified file name. The buffer must be distinct from the *partialname*, *override-name*, *search*, and *defaults* areas.

maxlen is the length in bytes of the string variable *fullname*.

fullname-length output

INT .EXT:ref:1

returns the actual length of the resolved file name returned in *fullname*. 0 is returned if an error occurs.

options input

INT:value

specifies options. If omitted, the default value is 0. The bits, when equal to 1, indicate:

Bit	Meaning
<0:7>	Reserved (specify 0).
<8>	If *partialname* consists of a simple unqualified disk file name, a DEFINE name is generated to use as the *override-name*. The generated name is an equal sign (=) followed by *partialname*. This name convention corresponds to the ASSIGN name convention used by TAL. This option cannot be used unless *override-name* is omitted or has a length of 0.
<9>	If *search* is supplied and a search fails to find an existing file, FILENAME_RESOLVE_ resolves *partialname* using the first entry in the search DEFINE. If this option is not specified and a search fails to find an existing file, FILENAME_RESOLVE_ returns error 11.
<10>	If a DEFINE name other than one translated by *options*.<11> or *options*.<12> is supplied for *partialname*, FILENAME_RESOLVE_ returns error 13.

Bit	Meaning
<11>	If a DEFINE name is supplied for *partialname*, and if the DEFINE has a file name associated with it, that file name is returned as the result. (Error 198 is returned if the DEFINE doesn't exist; error 13 is returned if DEFMODE is OFF.) If neither this option nor *options*.<12> is specified, then FILENAME_RESOLVE_ returns the DEFINE name as the result. This option causes DEFINE names to be translated more often than *options*.<12>, but doing so causes the extra information carried in the DEFINEs to be lost. (Note that CLASS TAPE and SPOOL DEFINEs carry such information. TAPE DEFINEs do not necessarily have any specific file name associated with them, and so are not always translated by this option. CLASS SORT, SUBSORT, CATALOG, DEFAULTS, and SEARCH DEFINE names are never changed by this option.)
<12>	If a DEFINE name is supplied for *partialname*, and if the DEFINE contains only a file name (that is, it is a simple MAP DEFINE), then FILENAME_RESOLVE_ returns that file name as the result. (Error 198 is returned if the DEFINE doesn't exist; error 13 is returned if DEFMODE is OFF.) If neither this option nor *options*.<11> is specified, then the DEFINE name is returned as the result.
<13>	If a logical device number (LDEV) appears as part of *partialname*, FILENAME_RESOLVE_ translates it to the corresponding symbolic device name.
<14>	A single name part supplied in *partialname* is to be treated as a subvolume name or pattern.
<15>	All alphabetic characters in the resolved file name are to be shifted to upper case. If this option is not specified, characters transferred from *partialname* to *fullname* are unchanged. Characters taken from *defaults* are always shifted to upper case, regardless of this option.

override-name:*length* input:input

STRING .EXT:ref:*, INT:value

if supplied and if *length* is not 0, specifies a DEFINE name to be used as the primary input instead of *partialname*. If used, the value of *override-name* must be exactly *length* bytes long. If the DEFINE name does not exist, or if DEFMODE is OFF, *partialname* is processed as normal.

search:*length* input:input

STRING .EXT:ref:*, INT:value

if supplied and if *length* is not 0, specifies the name of a CLASS SEARCH DEFINE that is used to resolve single-part file names by testing for the file's existence in several subvolumes. See "Considerations" for details. If used, the value of *search* must be exactly *length* bytes long. If the DEFINE does not exist, DEFMODE is OFF, or *partialname* does not consist of a single name part, then no searching is done and no error is reported.

defaults:*length* input:input

STRING .EXT:ref:*, INT:value

if supplied and if *length* is not 0, specifies either a default subvolume to be used for name resolution or the name of a CLASS DEFAULTS DEFINE to be used for name resolution. If used, the value of *defaults* must be exactly *length* bytes long and must be in this form:

[[*node*.]$*volume*.]*subvolume*

Any part of the supplied default value that is used in the resolved file name is shifted to upper case, regardless of the value of *options*.<15>.

Omitted name parts are taken from the =_DEFAULTS DEFINE. If this parameter is omitted or if *length* is 0, the value of the VOLUME attribute of the =_DEFAULTS DEFINE is used.

Considerations

△ **Caution.** Passing an invalid file name or file-name pattern to this procedure can result in a trap, a signal, or data corruption. To verify that a file name or file-name pattern is valid, use the FILENAME_SCAN_ procedure.

- FILENAME_RESOLVE_ performs the principal steps of its operation in this order.
  1. If the caller supplied the name of an existing DEFINE in *override-name*, it substitutes the DEFINE name for *partialname*.
  2. If the criteria for doing a search are met, it performs a search.
  3. If the caller specified the appropriate options, it resolves or reduces DEFINEs.
  4. It applies default values and resolves LDEVs.
  5. It changes the result to upper case.
- *options*.<14> specifies that a single name part supplied in *partialname* be treated as a subvolume name or pattern. This means that the input string “f” is resolved to the form “\SYS.$VOL.f”. Without this option, “f” would be resolved to “\SYS.$VOL.SUBV.f”.
- The name $RECEIVE is never expanded with a node name.
- FILENAME_RESOLVE_ does not modify DEFINE names except to change them to upper case. It also does not check DEFINE names in *partialname* for existence, except as noted under *options* bits 11 and 12.
- If you are using FILENAME_RESOLVE_ to obtain a file name in standard form (to use as a key, for instance), then you should use *options* bits 13 (LDEV resolve) and 15 (upper case).
- Override name

The *override-name* parameter provides a way to allow interactive users to override a program's normal choice of file name with a CLASS MAP (or other) DEFINE. The programmer provides the default file name in *partial-name* and the DEFINE name that the interactive user would use in *override-name*.

options.<8> provides a way to automatically generate an override name from the default name if they are to have a direct correspondence.

- Search lists

  If *search* specifies an existing CLASS SEARCH DEFINE and DEFMODE is ON, FILENAME_RESOLVE_ can resolve file names by search. The SEARCH DEFINE contains a sequence of subvolumes to be inspected or names of CLASS DEFAULTS DEFINEs that contain subvolumes to be inspected.

  The search is performed if *partialname* consists of only the file identifier part of a file name, that is, if it does not include any delimiters or other special characters (such as . : $ * ?). The first subvolume in the DEFINE is checked for the existence of a file with the supplied name, and if that fails, each of the remaining subvolumes in the SEARCH DEFINE are similarly checked. If no such file exists in any of the subvolumes, FILENAME_RESOLVE_ returns error 11. (The exception to this is when *options*.<9> is equal to 1, in which case FILENAME_RESOLVE_ returns the name as resolved using the first entry in the SEARCH DEFINE.) If a file is found, its complete name is returned in *fullname*.

  If either the *search* parameter specifies a DEFINE that is not CLASS SEARCH or the *defaults* parameter specifies a DEFINE that is not CLASS DEFAULTS, an error 113 is returned.

Example

In this example, `name` is resolved using a search list if the DEFINE =SRCHLST exists; if the DEFINE does not exist, `name` is resolved by normal means.

```
slist ':=' "=SRCHLST";
error:= FILENAME_RESOLVE_( name:namelen,
                           outname:256, outnamelen, , ,
                           slist:8 );
```

Related Programming Manual

For programming information about the FILENAME_RESOLVE_ procedure, see the *Guardian Programmer's Guide*.

FILENAME_SCAN_ Procedure

Summary

The FILENAME_SCAN_ procedure checks for valid file-name syntax and returns the length in bytes of that part of the input string that constitutes a file name. Node names are accepted as valid input, as are partially or fully qualified names of disk files, processes, and devices. File-name patterns and subvolume names are accepted when you select the appropriate options. FILENAME_SCAN_ checks syntax only; no check for the existence of any entity is performed.

For the definitions of file name and file-name pattern, see Appendix D, File Names and Process Identifiers.

Syntax for C Programmers

```
#include <cextdecs(FILENAME_SCAN_)>

short FILENAME_SCAN_ ( const char *string
                      ,short length
                      ,[ short *count ]
                      ,[ short *kind ]
                      ,[ short *entity-level ]
                      ,[ short options ] );
```

Syntax for TAL Programmers

```
error := FILENAME_SCAN_( string:length     ! i:i
                        ,[ count ]         ! o
                        ,[ kind ]          ! o
                        ,[ entity-level ]  ! o
                        ,[ options ] );    ! i
```

Parameters

error returned value

INT

is a file-system error number indicating the outcome of the operation. Possible values include:

0	Syntactically correct name found for *count* bytes; see description of *count* parameter
13	The form of the name found is incorrect; also, various program and resource errors

string:*length* input:input

STRING .EXT:ref:*, INT:value

is a character string to be searched for a valid file name. *string* must be exactly *length* bytes long. A valid file name must begin at the first character of *string*. It can occupy the entire length of *string*, or it can occupy the left-hand portion of *string* and be followed by characters that cannot appear in a valid file name or file-name pattern.

count output

INT .EXT:ref:1

is the number of characters occupied by the name if a valid name is found. If error 13 is returned in *error*, *count* contains the number of characters examined when the name was determined to be invalid. To know that the entire input string constitutes a valid name, you should verify that *count* is equal to *length*. See Example on page 5-202.

kind output

INT .EXT:ref:1

identifies the class of name that was found. Possible values are:

0	File name (that is, the name of an entity).
1	File-name pattern. This value can be returned only if *options*.<15> is equal to 1. If the input is a name as well as a file-name pattern (that is, it does not contain an asterisk or question mark), FILENAME_SCAN_ classifies it as a name and not a pattern.
2	DEFINE name.

entity-level output

INT .EXT:ref:1

identifies the class of entity represented by the supplied name based on its syntax. This value corresponds to the “level” of the rightmost name section that appears. Possible values are:

-1 Node name.
0 Name of device or process without qualifiers.
> 0 Name of device or process with *entity-level* qualifiers.

Note that a disk file name always has an *entity-level* greater than 0 (1 if it is a temporary file; 2 if it is a permanent file). The value returned is not affected by whether name sections to the left are implied (that is, defaulted).

A DEFINE name always yields a value of 0.

options input

INT:value

specifies options. The bits, when equal to 1, have these meanings:

<0:13> Reserved (specify 0).

<14> Specifies that a subvolume name be accepted as valid input.

<15> Specifies that a file-name pattern be accepted as valid input.

The default value is 0.

Considerations

- The syntax checking performed by FILENAME_SCAN_ includes checks that the lengths of individual name parts are acceptable. (For example, it checks that a subvolume name is no more than 8 characters long.) Unless a name occupies the entire input string, FILENAME_SCAN_ also requires that the character following the name must not belong to the set of characters that can appear in a file name or file-name pattern.
- FILENAME_SCAN_ checks only that some left-hand part of the input string is a valid name; it does not require that the name occupy the entire string. If you need such a check, you can compare the length of *string* to the returned *count*.

Example

This example checks that name is a valid file name with a length of namelen bytes.

```
error := FILENAME_SCAN_ ( name:namelen, count );
IF error <> 0 OR count <> namelen THEN  ! if bad name then
   CALL BAD^FILENAME^FOUND;             ! call user procedure
```

Related Programming Manual

For programming information about the FILENAME_SCAN_ procedure, see the *Guardian Programmer's Guide*.

FILENAME_TO_OLDFILENAME_ Procedure

Summary

The FILENAME_TO_OLDFILENAME_ procedure converts a file name to the C-series internal file-name format. See Appendix D, File Names and Process Identifiers for descriptions of C-series and D-series file names.

Syntax for C Programmers

```
#include <cextdecs(FILENAME_TO_OLDFILENAME_)>

short FILENAME_TO_OLDFILENAME_ ( const char *filename
                                ,short length
                                ,short *oldstyle-name );
```

Syntax for TAL Programmers

```
error := FILENAME_TO_OLDFILENAME_ ( filename:length     ! i:i
                                   ,oldstyle-name );    ! o
```

Parameters

error returned value

INT

returns a file-system error number indicating the outcome of the operation.

filename:*length* input:input

STRING .EXT:ref:*, INT:value

specifies the valid file name to be converted. The value of *filename* must be exactly *length* bytes long. If the name is partially qualified, it is resolved using

the contents of the user's =_DEFAULTS DEFINE. See caution under "Considerations."

oldstyle-name output

INT .EXT:ref:12

returns the internal-format file name.

Considerations

△ **Caution.** Passing an invalid file name to this procedure can result in a trap, a signal, or data corruption. To verify that a file name is valid, use the FILENAME_SCAN_ procedure.

- The process file name of an unnamed process can be converted if it has a PIN of 255 or less.
- If *filename* contains a node name or if the default node name is remote, *oldstyle-name* is normally returned in internal network form. Otherwise, *oldstyle-name* is in internal local form.

  An exception occurs when an 8-character destination name (for example, "$LONGDEV") is supplied as part of *filename*. Such a name is converted without error into internal local form if the local node is explicitly designated in *filename* or in the =_DEFAULTS DEFINE (if *filename* does not contain a node name). Otherwise error 20 is returned.

Example

```
error := FILENAME_TO_OLDFILENAME_ ( fname:length,
                                    oldstylename );
```

Related Programming Manual

For programming information about the FILENAME_TO_OLDFILENAME_ procedure, see the *Guardian Application Conversion Guide*.

FILENAME_TO_PATHNAME_ Procedure

Summary

The FILENAME_TO_PATHNAME_ procedure converts a Guardian file name or subvolume name to an OSS pathname. See Appendix D, File Names and Process Identifiers for a descriptions of OSS pathname syntax.

Syntax for C Programmers

```
#include <cextdecs(FILENAME_TO_PATHNAME_)>

short FILENAME_TO_PATHNAME_ ( const char *filename
                             ,short length
                             ,char *pathname
                             ,short maxlen
                             ,short *pathlen
                             ,short [ options ]
                             ,short [ *index ] );
```

Syntax for TAL Programmers

```
error := FILENAME_TO_PATHNAME_ ( filename:length     ! i:i
                                ,pathname:maxlen     ! o:i
                                ,pathlen             ! o
                                ,[ options ]         ! i
                                ,[ index ] );        ! i,o
```

Parameters

error returned value

INT

returns a file-system error number indicating the outcome of the operation.

0 No error.

563 The *pathname* buffer is too small to contain the resulting name.

4002 *filename* specifies a Guardian name for an OSS file that either does not exist or has been unlinked but is still open by some process. The corresponding OSS errno value is ENOENT.

4006 The fileset that corresponds to the supplied Guardian name for an OSS file is not mounted. The corresponding OSS errno value is ENXIO.

4013 The caller does not have search access to one of the directories within all of the resulting pathnames. The corresponding OSS errno value is EACCESS.

4014 A parameter has an invalid address. The corresponding OSS errno value is EFAULT.

4022 One of these conditions has occurred: *options* is specified and *options*.<0:12> does not contain all zeros; *filename* is not a valid file or subvolume name; when resolving multiple pathames to a file, *index* does not correspond to a pathname of the file; or *options*.<13> = 0, *options*.14 = 1 and a `chroot()` function was executed which changed the root to something other than "/." The corresponding OSS `errno` value is `EINVAL`.

4202 The root fileset is not mounted. The corresponding OSS `errno` value is `ENOROOT`.

4211 The resulting *pathname* is longer than the limit defined in PATH_MAX. (`PATH_MAX` is a symbolic constant defined in the OSS `limits.h` header file.) The corresponding OSS `errno` value is `ECWDTOOLONG`.

filename:*length* input:input

STRING .EXT:ref:*, INT:value

specifies the name of the file or subvolume to be converted. To indicate that *filename* contains a subvolume name, use the *options* parameter. The value of *filename* must be exactly *length* bytes long, and it must be a valid disk file name. If the name is partially qualified, it is resolved using the contents of the VOLUME attribute of the =_DEFAULTS DEFINE.

pathname:*maxlen* output:input

STRING .EXT:ref:*, INT:value

returns the null-terminated OSS pathname that corresponds to the Guardian *filename*. *maxlen* specifies the maximum length in bytes of *pathname*, including the terminating null character.

pathlen output

INT .EXT:ref:1

returns the actual length in bytes of the *pathname* parameter, including the terminating null character.

options input

INT:value

specifies options for the *filename* parameter:

`<0:12>` Reserved (specify 0).

`<13>` 1 If the caller has appropriate privileges, specifies that *pathname* is an absolute pathname with respect to the system root.

0 Specifies that *pathname* is an absolute pathname with respect to the current root of the process.

<14>	1	Specifies that *pathname* always includes the system name in the form /E/system/path.
	0	Specifies that *pathname* includes system names only for remote pathnames; that is, local pathnames do not start with "/E."
<15>	1	Specifies that a subvolume name be accepted as valid input.
	0	Specifies that a the input must be a file name.

The default value is 0.

index input,output

INT .EXT:ref:1

specifies the index of the link to the named file to be returned in *pathname*. Specifying a value of -1 finds only the first accessible path. Specifying a value of 0 starts a search for all possible pathnames.

index returns the index to the next pathname to the file. A value of -1 on return indicates that *pathname* contains the last (or only) name for the file.

The default value is -1.

OSS Considerations

- If the supplied Guardian file name is not the Guardian file name of an OSS file, *pathname* returns an absolute pathname of the form:

  /G[/*volume*[/*subvolume*[/*file-id*]]]

  - The number of components in the pathname is the same as the number in the Guardian file name plus the /G prefix.
  - *volume*, if present, is derived from the Guardian volume name by removing the dollar sign ($).
  - *subvolume*, if present, is the Guardian subvolume name, including any preceding pound sign (#).
  - If the file name contains a node name, it must be that of the node on which the procedure is called.
  - Periods (.) in the Guardian file name are replaced by slashes in the pathname.
  - Alphabetic characters in the pathname are all lower case except for the "G" in "/G."
- If the supplied Guardian file name is the Guardian file name of an OSS file, *pathname* returns the corresponding absolute pathname of the OSS file.
- Some OSS files can have multiple pathnames because additional directory entries (or links) can be created for existing files. The pathname returned is the first one found for which the caller has search access unless the *index* parameter is used.

If *index* is used, all the file names can be returned by making multiple calls to FILENAME_TO_PATHNAME_ and using the value returned in *index* from each call as the value supplied in *index* for the next call. All pathnames have been returned when the returned value of *index* is -1.

- An error is returned (EINVAL) if index does not correspond to a pathname of the file; for example, as a result of unlinking a pathname between iterations of the FILENAME_TO_PATHNAME_ procedure. This error condition also indicates that there are no further pathnames to this file because index values are always contiguous.
- Two additional file numbers might be allocated: one for the OSS root directory and one for the OSS current working directory. These files are not necessarily the next available file numbers and they cannot be closed by calling FILE_CLOSE_.
- A current working directory is established from the value of the VOLUME attribute of the =_DEFAULTS DEFINE.
- The resident memory used by the calling process increases by a small amount.
- If the resulting pathname represents a file in the Guardian name space (/G), then the system does not check whether such a file exists.

Example in C

```
ret = FILENAME_TO_PATHNAME_(argv[1], /* Guardian file name */
      (short)strlen(argv[1]),        /* length of file name */
      pathname,                      /* buffer for OSS path
                                        name */
      PATH_MAX,                      /* length of buffer */
      &pathlen,                      /* length of path name */
      , );
```

Related Programming Manual

For programming information about the FILENAME_TO_PATHNAME_ procedure, see the *Open System Services Programmer's Guide*.

FILENAME_TO_PROCESSHANDLE_ Procedure

Summary

The FILENAME_TO_PROCESSHANDLE_ procedure converts a file name to a process handle.

Syntax for C Programmers

```
#include <cextdecs(FILENAME_TO_PROCESSHANDLE_)>

short FILENAME_TO_PROCESSHANDLE_ ( const char *filename
                                  ,short length
                                  ,short *processhandle );
```

Syntax for TAL Programmers

```
error := FILENAME_TO_PROCESSHANDLE_ ( filename:length    !
i:i
                                     ,processhandle );   ! o
```

Parameters

error returned value

INT

returns a file-system error number indicating the outcome of the operation.

filename:*length* input:input

STRING .EXT:ref:*, INT:value

contains the valid process file name to be translated. The value of *filename* must be exactly *length* bytes long. If qualifiers are present, they are ignored. If a node name is not present, the current default node name in the =_DEFAULTS DEFINE is used. See caution under “Considerations.”

processhandle output

INT .EXT:ref:10

returns the process handle of the process designated by *filename*.

Considerations

△ **Caution.** Passing an invalid file name to this procedure can result in a trap, a signal, or data corruption. To verify that a file name is valid, use the FILENAME_SCAN_ procedure.

- If the file name to be converted by FILENAME_TO_PROCESSHANDLE_ designates something besides a process (for example, a disk file or a tape device),

the procedure returns the process handle of the process that controls the device (that is, the I/O process).

- When converting the process file name of a named process, FILENAME_TO_PROCESSHANDLE_ looks up the process by name in the destination control table (DCT). If the name is not found, error 14 is returned. However, it is sometimes possible for the name of a nonexistent process to be found in the DCT, in which case error 0 is returned. Therefore, even for a named process, error 0 (successful conversion of a process handle) does not guarantee that the process exists.

Related Programming Manual

For programming information about the FILENAME_TO_PROCESSHANDLE_ procedure, see the *Guardian Programmer's Guide*.

FILENAME_UNRESOLVE_ Procedure

Summary

The FILENAME_UNRESOLVE_ procedure accepts a file name as input, deletes left-hand sections that match the default values, and returns a file name that is semantically equivalent to the input file name.

Syntax for C Programmers

```
#include <cextdecs(FILENAME_UNRESOLVE_)>

short FILENAME_UNRESOLVE_ ( const char *longname
                           ,short length
                           ,char *shortname
                           ,short maxlen
                           ,short *shortname-length
                           ,[ short level ]
                           ,[ const char *defaults ]
                           ,[ short length ] );
```

- The character-string parameters *longname* and *defaults* are each followed by a parameter *length* that specifies the length in bytes of the character string. Where

the parameters are optional, the character-string parameter and the corresponding length parameter must either both be supplied or both be absent.

Syntax for TAL Programmers

```
error := FILENAME_UNRESOLVE_ ( longname:length            !
i:i
                              ,shortname:maxlen           !
o:i
                              ,shortname-length           ! o
                              ,[ level ]                  ! i
                              ,[ defaults:length ] );     !
i:i
```

Parameters

error returned value

INT

is a file-system error number indicating the outcome of the operation.

longname:*length* input:input

STRING .EXT:ref:*, INT:value

contains the valid file name or file-name pattern to be acted upon by FILENAME_UNRESOLVE_. The value of *longname* must be exactly *length* bytes long. See caution under “Considerations.”

shortname:*maxlen* output:input

STRING .EXT:ref:*, INT:value

defines the buffer where the resultant file name is to be placed. This buffer can occupy the same area as *longname*. The length of the resultant name is never greater than the length of *longname*.

maxlen is the length in bytes of the string variable *shortname*.

shortname-length output

INT .EXT:ref:1

returns the length in bytes of the name returned in *shortname*. If an error occurs, 0 is returned.

level input

INT:value

specifies the first part of the file name, scanning from the left, that should be returned even if it matches the default name. Name parts of this level and greater

are always returned if they are present in *longname*. If omitted, the default level is 0 (that is, no more than the node name is to be removed). Valid values are:

-1 Node name
0 Destination name (for example, volume, device, or process)
1 First qualifier (for example, subvolume)
2 Second qualifier (file identifier if disk file)

defaults:*length* input:input

STRING .EXT:ref:*, INT:value

if supplied and if *length* is not 0, specifies either a subvolume name to be used as the default subvolume name or the name of a CLASS DEFAULTS DEFINE. The contents of *defaults* are compared with *longname* to perform the unresolved operation.

If used, the value of *defaults* must be exactly *length* bytes long and must be in this form:

[[*node*.]$*volume*.]*subvolume*

Omitted name parts are taken from the =_DEFAULTS DEFINE.

If this parameter is omitted or if *length* is 0, the value of the VOLUME attribute of the =_DEFAULTS DEFINE is used.

Considerations

△ **Caution.** Passing an invalid file name or file-name pattern to this procedure can result in a trap, a signal, or data corruption. To verify that a file name or file-name pattern is valid, use the FILENAME_SCAN_ procedure.

- The FILENAME_UNRESOLVE_ procedure compares a specified file name with the default subvolume specification and removes left-hand sections that are identical. It scans the input file name from the left, and when it finds a difference, it returns that part and everything to the right. Name parts are never removed from the section indicated by *level* or from sections to the right of that point.

Examples

```
error := FILENAME_UNRESOLVE_ ( longname:longlen,
                               shortname:maxlen,
```

```
                                shortname-len,
                                level );
```

This table gives some possible input values for the above example, along with the output. Assume that the current default values are "\SYS.$VOL.SUB".

longname (input)	***level***	***shortname*** (output)
\mysys.$myvol.mysvol.myfile	0	\mysys.$myvol.mysvol.myfile
\sys.$myvol.mysvol.myfile	0	$myvol.mysvol.myfile
\sys.$vol.mysvol.myfile	0	$vol.mysvol.myfile
mysvol.myfile	0	mysvol.myfile
sub.myfile	1	sub.myfile
sub.myfile	2	myfile

Related Programming Manual

For programming information about the FILENAME_UNRESOLVE_ procedure, see the *Guardian Programmer's Guide*.

FILERECINFO Procedure (Superseded by FILE_GETINFOLISTBYNAME_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The FILERECINFO procedure obtains record characteristics of a disk file.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
CALL FILERECINFO ( [ filenum ]                              ! i
                 ,[ current-keyspecifier ]                  ! o
                 ,[ current-keyvalue ]                      ! o
                 ,[ current-keylen ]                        ! o
                 ,[ current-primary-keyvalue ]              ! o
                 ,[ current-primary-keylen ]                ! o
                 ,[ partition-in-error ]                    ! o
                 ,[ specifier-of-key-in-error ]             ! o
                 ,[ file-type ]                             ! o
                 ,[ logical-recordlen ]                     ! o
                 ,[ blocklen ]                              ! o
                 ,[ key-sequenced-parameters ]              ! o
                 ,[ alternate-key-parameters ]              ! o
                 ,[ partition-parameters ]                  ! o
                 ,[ file-name ] );                          ! i
```

Parameters

filenum input

INT:value

is the number of an open file that identifies the file whose characteristics are to be returned. You must specify either *filenum* or *file-name*; specifying both causes a CCL condition code.

current-keyspecifier output

INT:ref:1

returns the current key field's key specifier. This is invalid when you specify the *file-name* parameter; use *filenum*.

current-keyvalue output

STRING:ref:*

returns the value of the current key for *current-keylen* bytes. This is invalid when you specify the *file-name* parameter; use *filenum*. This value is not valid for queue files. Also, this parameter cannot be used with a non-key-sequenced file opened with 64-bit primary keys open flag. If an attempt is made, the call will fail with condition code CCL.

current-keylen output

INT:ref:1

returns the current key length in bytes. This is invalid when the *file-name* parameter is specified; use *filenum*.

current-primary-keyvalue output

STRING:ref:*

returns the value of the current primary key for *current-primary-keylen* bytes. This is invalid when you specify the *file-name* parameter; use *filenum*. This parameter cannot be used with a non-key-sequenced file opened with a 64-bit primary keys open flag. If an attempt is made, the call will fail with condition code CCL.

current-primary-keylen output

INT:ref:1

returns the length, in bytes, of the current primary key. This is invalid when you specify the *file-name* parameter; use *filenum*.

partition-in-error output

INT:ref:1

returns a number from 0 through 15 that indicates the partition in which the latest error occurred for this file. This is invalid when you specify the *file-name* parameter; use *filenum*.

specifier-of-key-in-error output

INT:ref:1

returns the key specifier associated with the latest error occurring with this file. This is invalid when you specify the *file-name* parameter; use *filenum*.

These parameters are the only parameters returned when you specify *file-name*:

file-type output

INT:ref:1

returns a number indicating the type of file being accessed.

<2> 1 For systems with the Transaction Management Facility, indicates this file is audited.

<5:7> Specifies object type for SQL object file:

0 File is not SQL
2 File is an SQL table
4 File is an SQL index

5 File is an SQL protection view
7 File is an SQL shorthand view

<9> 1 Specifies that this is a queue file.

<10> 1 Means REFRESH is specified for this file.

<11> 1 For key-sequenced files, means index compression is specified.

<12> 1 For key-sequenced files, means data compression is specified.

1 For unstructured files, means ODDUNSTR is specified.

<13:15> Specifies the file structure:

0 Unstructured
1 Relative
2 Entry-sequenced
3 Key-sequenced

logical-recordlen output

INT:ref:1

returns the maximum size of the logical record in bytes.

blocklen output

INT:ref:1

returns the length, in bytes, of a block of records for the file.

key-sequenced-parameters output

INT:ref:*

is an array where the parameters unique to a key-sequenced file are returned. (For the format of this array, see CREATE Procedure (Superseded by FILE_CREATELIST_ Procedure).)

alternate-key-parameters output

INT:ref:*

is an array where the parameters describing the file's alternate keys are returned. (For the format of this array, see CREATE Procedure (Superseded by FILE_CREATELIST_ Procedure). The length of the array can be obtained by calling FILEINQUIRE.)

partition-parameters output

INT:ref:*

is an array where the parameters describing a multivolume file are returned. (For the format of this array, see CREATE Procedure

(Superseded by FILE_CREATELIST_ Procedure). The length of the array can be obtained by calling FILEINQUIRE.) The 2-byte unsigned extent size fields of this parameter are cannot represent all possible values. When a value is not representable, -1 is substituted. The superseding procedure must be used to get the correct value.

file-name input

INT:ref:12

is an internal-format file name that identifies the file whose characteristics are returned. You must specify either *filenum* or *file-name*; specifying both causes a CCL condition code.

When you specify *file-name*, the only parameters returned are *filetype*, *logical-recordlen*, *blocklen*, *key-sequenced-parameters*, *alternate-key-parameters*, and *partition-parameters*.

This information is acquired from the volume directory and not from any system control structures, so there is no check to see if the file is actually opened by this or any other process.

Considerations

- The FILERECINFO procedure is used to determine whether a file is a queue file or an ordinary key-sequenced file. This procedure should not be used for determining the value of the current key of the queue file, because the current key position is not maintained for queue files. The *current-keyvalue* parameter that would be returned for a queue file is undefined.

Condition Codes

< (CCL) indicates that an error occurred. This can indicate that the specified file was not found or that both *filenum* and *file-name* were specified in the same FILERECINFO call. This can also indicate that the *current-keyvalue* and *current-primary-keyvalue* parameters were used with non-key-sequenced files opened with 64-bit primary keys open flag.

= (CCE) indicates that FILERECINFO executed successfully.

> (CCG) indicates that the file is not a disk file.

Example

```
CALL FILERECINFO ( FILE^NUMBER
                 ,            ! current key specifier.
                 ,            ! current key value.
                 ,            ! current key length.
                 ,            ! current primary key value.
                 ,            ! current primary key length.
```

```
,             ! partition in error.
,             ! key in error.
,FILE^TYPE );
```

FIXSTRING Procedure

Summary

The FIXSTRING procedure is used to edit a string based on subcommands provided in a template.

Syntax for C Programmers

```
#include <cextdecs(FIXSTRING)>

_cc_status FIXSTRING ( char *template
                      ,short template-len
                      ,char *data
                      ,short _near *data-len
                      ,[ short maximum-data-len ]
                      ,[ short _near *modification-status ]
);
```

- The function value returned by, which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL FIXSTRING ( template                          ! i
                ,template-len                      ! i
                ,data                              ! i,o
                ,data-len                          ! i,o
                ,[ maximum-data-len ]              ! i
                ,[ modification-status ] );        ! o
```

Parameters

template input

STRING:ref:*

is the character string to be used as a modification template.

There are three basic subcommands that you can use in *template*: replacement, insertion, and deletion.

In addition, replacement can be either explicit (a subcommand beginning with “R”) or implicit (a subcommand beginning with any nonblank character other than “R,” “I,” or “D”). The form of *template* is:

```
template = { subcommand // ... }

subcommand =

   { Rreplacement string }    ! replace subcommand
   { Iinsertion string   }    ! insert subcommand
   { D                   }    ! delete subcommand
   { replacement string  }    ! implicit replacement
```

template-len input

INT:value

is the length, in bytes, of the template string.

data input, output

STRING:ref:*

on input, is a string to be modified. The resulting string returns in this parameter.

data-len input, output

INT:ref:1

on input, contains the length, in bytes, of the string input in *data*. On return, it contains the length, in bytes, of the modified data string in *data*.

maximum-data-len input

INT:value

contains the maximum length, in bytes, to which *data* can expand during the call to FIXSTRING. If omitted, 132 is used for this value.

modification-status output

INT:ref:1

returns an integer value as follows:

0 No change was made to *data*.
1 A replacement, insertion, or deletion was performed on *data* (see "Considerations").

Condition Code Settings

< (CCL) indicates that one or more of the required parameters is missing.

= (CCE) indicates that the operation completed successfully.

> (CCG) indicates that an insert or replace would have caused the *data* string to exceed the *maximum-data-len*.

Considerations

- *template* considerations

  A character in *template* is recognized as the beginning of a subcommand if it is the first nonblank character in *template*, the first nonblank character following "//," or the first nonblank character following a "D" subcommand. Otherwise, it is considered part of a previous subcommand.

  Note that a subcommand may immediately follow "D" without being preceded by "//."

  If a subcommand begins with "R," "I," or "D," it is recognized as an explicit command. Otherwise, it is recognized as an implied replacement.

  The action of the subcommands is as follows:

  - R (or r) for "replace"

    This subcommand replaces characters in *data* with *replacement-string* on a one-for-one basis. Replacement begins with the character corresponding to R. The *replacement-string* is terminated by the end of *template* or by a "//" sequence in *template*. Trailing blanks are considered part of the replacement string (that is, blanks are not ignored).

  - Implied replacement

    A subcommand that does not begin with "R," "I," or "D" is recognized as a *replacement-string*. Characters in *replacement-string* replace the corresponding characters in *data* on a one-for-one basis.

  - D (or d) for "delete"

    This subcommand deletes the corresponding character in *data*.

  - I (or i) for "insert"

This subcommand inserts a string from *template* into *data* preceding the character corresponding to the "I". The *insertion-string* is terminated by the end of *template* or by a "//" sequence in *template*. Trailing blanks are considered part of the insertion string (that is, they are not ignored).

- When *data* is truncated

  The *maximum-data-len* serves to protect data residing past the end of the *data* string. Therefore, *data* is truncated whenever *data-len* exceeds *maximum-data-len* during processing by FIXSTRING.

  In particular, FIXSTRING truncates *data* if *data-len* temporarily exceeds *maximum-data-len*, even if *template* contains delete subcommands that result in a *data* string of the correct length.

- When insertion string is truncated

  If an insertion causes the length of *data* to exceed *maximum-data-len*, the FIXSTRING truncates *insertion-string*.

- *modification-status* is equal to 1 if a replacement is performed that leaves *data* unchanged.

Example

```
CALL FIXSTRING ( S^TEMP^ARRAY , TEMP^LEN , SCOMMAND , NUM );
```

Related Programming Manual

For programming information about the FIXSTRING utility procedure, see the *Guardian Programmer's Guide*.

FNAME32COLLAPSE Procedure (Superseded)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

FNAME32COLLAPSE converts the 32-character file name used by the Distributed Name Service to external format for display.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
length := FNAME32COLLAPSE ( intname        ! i
                          ,extname )       ! o
```

Parameters

length returned value

INT

is the number of bytes in *extname* or 0 if an error occurred.

intname input

STRING .EXT:ref:32

is a 32-character array containing a subsystem object name in the form:

```
\sysname$volume subvol  file-id
```

extname output

STRING .EXT:ref:35

contains, on return, the external form of *intname*:

```
\sysname.$volume.subvol.file-id
```

Considerations

- The caller must pass a valid subsystem object name in *intname*. Invalid names cause unpredictable results.
- If a parameter is missing or a bounds error occurs on a parameter, *length* will contain 0.

Related Programming Manual

For network programming applications, see the *Distributed Name Service (DNS) Manual*.

FNAME32EXPAND Procedure (Superseded by FILENAME_SCAN_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

FNAME32EXPAND expands a partial file name from the compacted external form to the 32-character file name used by the Distributed Name Service (DNS) programmatic interface.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
length := FNAME32EXPAND ( extname          ! i
                        , intname          ! o
                        , defaults );      ! i
```

Parameters

length returned value

INT

is the length, in bytes, of the file name in *extname*, or 0 if an error occurred.

extname input

STRING .EXT:ref:35

is the file name to be expanded. The file name must be in one of the forms acceptable to FNAMEEXPAND. For details, see FNAMEEXPAND Procedure

(Superseded by FILENAME_SCAN_ Procedure and FILENAME_RESOLVE_ Procedure).

intname output

STRING .EXT:ref:32

is an array of 32 characters where FNAME32EXPAND returns the expanded file name. This array can be the same array as *extname*.

defaults input

STRING .EXT:ref:16 or 18

is an array of eight words containing the default volume and subvolume name (and optionally system number) that is to be used in the file name expansion. This array has the same format as the corresponding parameter to FNAMEEXPAND.

Or it is an array of nine words where the first word contains the default system number and the remaining eight words contain the default volume and subvolume names.

Considerations

FNAME32EXPAND differs from FNAMEEXPAND in these ways:

- All 35 characters of the *extname* parameter must be addressable, even if the actual file name occupies less space.
- All alphabetic characters in the internal name are in upper case.
- Internal names returned by the procedure are always in network form.
- FNAME32EXPAND accepts file names that have eight-character device names in network format or file names where a default system number is passed in the *defaults* parameter.
- FNAME32EXPAND returns 0 if the *defaults* parameter specifies a system number that is not currently defined.
- FNAME32EXPAND returns 0 if a parameter is missing or if a bounds error occurs on a parameter. It also returns 0 if the first byte of *defaults* is not “\$”, blank, or 0.
- If a default system other than the caller’s system and an eight-character default volume name are desired, the *defaults* parameter must be in the nine-word format. FNAME32EXPAND interprets the *defaults* parameter as being nine words in length if the high-order byte of the first word is zero.

Related Programming Manual

For network programming applications, see the *Distributed Name Service (DNS) Manual*.

FNAME32TOFNAME Procedure (Superseded)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

FNAME32TOFNAME converts a file name from the 32-character format used by the Distributed Name Service (DNS) to its internal format.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
status := FNAME32TOFNAME ( fname32          ! i
                         ,fname );          ! o
```

Parameters

status returned value

INT

indicates the outcome of the call:

-1	File name successfully converted
0	File name cannot be converted, or an error occurred

fname32 input

STRING .EXT:ref:32

is the name to be converted. The array must contain a file name in DNS format.

fname output

STRING .EXT:ref:24

is a 24-character array where the name is returned.

Considerations

- If a parameter is missing or if a bounds error occurs on a parameter, 0 is returned.
- If the first eight characters of *fname32* contain the name of the system on which the procedure is called, *fname* is returned in local format.
- If the first eight characters of *fname32* name a system other than the one on which the procedure is called **and** the next eight characters specify an eight-character device name, FNAME32TOFNAME returns 0 to indicate that the file name cannot be converted to internal format because it is too long.
- It is the calling program's responsibility to pass a valid DNS file name in *fname32*. Invalid file names can cause unpredictable results.

Related Programming Manual

For network programming applications, see the *Distributed Name Service (DNS) Manual*.

FNAMECOLLAPSE Procedure (Superseded by OLDFILENAME_TO_FILENAME_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The FNAMECOLLAPSE procedure converts a file name from internal to external form. The system number of a network file name is converted to the corresponding system name.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
length := FNAMECOLLAPSE ( internal-name          ! i
                         ,external-name );       ! o
```

Parameters

length returned value

INT

returns the number of bytes in *external-name*.

internal-name input

INT:ref:12

is the name to be converted. *internal-name* is an array of 12 words. *internal-filename* cannot be the same array as *external-filename*. For a description of valid internal file names, see the discussion of "Internal File Names" in Appendix D, File Names and Process Identifiers.

external-name output

STRING:ref:26 or STRING:ref:34

returns the external form of *internal-name*. If *internal-name* is a local file name, *external-name* contains a maximum of 26 bytes; if a network name is converted, *external-name* contains a maximum of 34 bytes. (See the FNAMEEXPAND procedure.)

Considerations

- Invalid file names

  It is the responsibility of the program calling FNAMECOLLAPSE to pass a valid file name in *internal-name*. Invalid file names cause unpredictable results such as retrieving information from the wrong file.

- Passing a bad *sysnum* value

  If *internal-name* is in network form, and the system number in the second byte does not correspond to any system in the network, FNAMECOLLAPSE supplies "??" as the system name.

- System names as filenames

  The procedure does not always attach system names so that it will work properly as a filename. For example, the internal filename for an unnamed process produces a printable string; however, the string is not acceptable as a filename.

Example

```
LENGTH := FNAMECOLLAPSE ( INTNAME , EXTNAME );
```

Note. If INTNAME is passed in local internal form, for example "`$SYSTEM SUBVOL MYFILE`", it converts to the external local form "`$SYSTEM.SUBVOL.MYFILE`".

If INTNAME is passed in network form, for example "`\`*`sysnum`*`SYSTEMSUBVOL MYFILE`", it converts to the external network form, "`\`*`system-name`*`.$SYSTEM.SUBVOL.MYFILE`".

FNAMECOMPARE Procedure (Superseded by FILENAME_COMPARE_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The FNAMECOMPARE procedure compares two file names within a local or network environment to determine whether these file names refer to the same file or device. For example, one name might be a logical device number, while the other reference might be a symbolic name. The file names compared must be in the standard 12-word internal format that FNAMEEXPAND returns.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
status := FNAMECOMPARE ( filename1          ! i
                        ,filename2 );       ! i
```

Parameters

status returned value

INT

returns a value indicating the outcome of the comparison. Values for *status* are:

-1 The file names do not refer to the same file.

0 The file names refer to the same file.

1 The file names refer to the same volume name, device name, or process name on the same system; however, words [4:11] are not the same:

filename1[4] <> *filename2*[4] FOR 8

A value less than -1 is the negative of a file-system error code; in these cases, the comparison is not attempted.

filename1 input

INT:ref:12

is the first file name that is compared. Each *filename* array can contain either a local or a network file name in 12-word internal format. For the definitions of file names, see Appendix D, File Names and Process Identifiers.

filename2 input

INT:ref:12

is the second file name that is compared.

Considerations

- The arrays containing the file names for comparison are not modified.
- Alphabetic characters not upshifted

  Alphabetic characters within qualified process names are not upshifted before comparison.
- Passing DEFINE names

  Either or both of the file name parameters can be DEFINE names. For CLASS MAP DEFINEs, the procedure uses the file name given by the DEFINE to make the comparison. A name that designates a DEFINE of another class compares equal only to a name that designates the same DEFINE. If a DEFINE name is a

logical name but no such DEFINE exists, the procedure returns the negative file-system error -198 (missing DEFINE).

- Passing logical device numbers for file names

  If a logical device number format (such as $0076) is used for one file name but not for the second file name, the device table of the referenced system is consulted to determine whether the names are equivalent. This is the only case where the device table is used.

- FNAMECOMPARE and negative file errors

  Negative file-system error codes indicate that a logical device number format is passed for one file name and not for the second and that the device is connected to a remote network node. Some of the most common negative file-system error codes returned are:

  -13 An invalid file name specification for either file name is made.

  -14 The device does not exist. Only one of the file names is passed in logical device number format (requiring a check of the device table), and the file name represents a device connected to a remote node.

  -18 No such system is defined in this network. Only one of the file names is passed in logical device number format (requiring a check of the device table), and the file name represents a device connected to a remote node.

  -22 A parameter or buffer is out of bounds.

  -250 All paths to the system are down. Only one of the file names is passed in logical device number format (requiring a check of the device table), and the file name represents a device connected to a remote node.

Examples

```
FNAME1 ':=' [ "$TERM1" , 9 * [ "   "] ];
FNAME2 ':=' [ %56006 , "TERM1 " , 8 * [ "   "] ];
            ! "\ , "TERM1";
STATUS := FNAMECOMPARE ( FNAME1 , FNAME2 );
```

Execution of this example on system number 6 returns a 0 in STATUS.

On other systems, execution of the example returns a status of -1.

Whether a system is a network node or not, execution of

```
FNAME1 ':=' [ "$SERVR  #START  UPDATING" ];
FNAME2 ':=' [ "$SERVR  #FINISH UPDATING" ];
STATUS := FNAMECOMPARE ( FNAME1 , FNAME2 );
```

returns a status of +1.

In any system, execution of

```
FNAME1 ':=' [ "$0013 ", 9 * [ "   "] ];
FNAME2 ':=' [ "$DATAX", 9 * [ "   "] ];
STATUS := FNAMECOMPARE ( FNAME1 , FNAME2 );
```

returns a status of 0 if the device name $DATAX is defined as logical device number 13 at SYSGEN time; in all other cases, it returns a status of -1.

FNAMEEXPAND Procedure (Superseded by FILENAME_SCAN_ Procedure and FILENAME_RESOLVE_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The FNAMEEXPAND procedure is used to expand a partial file name from the compacted external form to the standard 12-word internal form usable by other file-system procedures.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
length := FNAMEEXPAND ( external-filename          ! i
                       ,internal-filename          ! o
                       ,default-names );           ! i
```

Parameters

length returned value

INT

returns the length, in bytes, of the file name in *external-filename*. If an invalid file name is specified, 0 is returned.

external-filename input

STRING:ref:27 or STRING:ref:35

is the file name to be expanded. The file name must be in the form:

[\sysname.]file-name or

definename

followed by a delimiter, and where *file-name* is in one of these forms:

[$*volname.][subvol-name.]file-id*

$*processname[.#1st-qualif-name[.2nd-qualif-name]]*

$*devname*

$*ldevnum*

the delimiter that follows *file-name* can be any character that is not valid as part of an external file name, such as blank or null. When the *external-filename* is 34 characters and a delimiter is required, the length of the *external-filename* expands to 35 characters.

When *system* is present, $*volname* cannot consist of eight characters unless *system* is the local system and *default-names* does not include a system number. In that case the output name is in local form.

internal-filename output

INT:ref:12

is an array of 12 words where FNAMEEXPAND returns the expanded file name. FNAMEEXPAND (unlike FNAMECOLLAPSE) can have the same source and destination buffers (file names) since it uses a temporary intermediate storage area for the conversion. (See “Considerations” for the form of the returned *internal-filename*.)

default-names input

INT:ref:8

is an array of eight words containing the default volume and subvolume names to be used in file name expansion. The *default-names* values are used when the corresponding values are not specified in *external-filename* (see “Considerations” below). *default-names* is of the form:

[0:3] default *volname*. First two bytes can be “*sysnum*,” in which case “$” is omitted from volume name. (blank-filled on right)

[4:7] default *subvolname* (blank-filled on right)

`[0:7]` corresponds directly to *word*[1:8] of the command interpreter startup message. For the startup message format, see the *Guardian Procedure Errors and Messages Manual*.

Considerations

- Expanding network file names

  FNAMEEXPAND converts local file names to local names and network file names to network names.

  When network file names are involved, FNAMEEXPAND converts the system name to the appropriate system number (see Example on page 5-234). (If the system name is unknown, FNAMEEXPAND supplies 255 for the system number; FNAMEEXPAND calls LOCATESYSTEM for this work.)

  Results of file name expansion by FNAMEEXPAND

  - *file-id* returns as:

    `[0:3]` $*default-volname* (blank-fill)
    `[4:7]` *default-subvolname* (blank-fill)
    `[8:11]` *file-id* (blank-fill)

  - *subvolname.file-id* returns as:

    `[0:3]` $*default-volname* (blank-fill)
    `[4:7]` *subvolname* (blank-fill)
    `[8:11]` *file-id* (blank-fill)

  - $*volname*.*file-id* returns as:

    `[0:3]` $*volname* (blank-fill)
    `[4:7]` *default-subvolname* (blank-fill)
    `[8:11]` *file-id* (blank-fill)

  - $*volname*.*subvolname*.*file-id* returns as:

    `[0:3]` $*volname* (blank-fill)
    `[4:7]` *subvolname* (blank-fill)
    `[8:11]` *file-id* (blank-fill)

  - $*processname*.#*1st-qualif-name* returns as:

    `[0:3]` *$processname* (blank-fill)
    `[4:7]` *#1st-qualif-name* (blank-fill)
    `[8:11]` (blank-fill)

  - $*processname.#1st-qualif-name.2nd-qualif-name* returns as:

    `[0:3]` *$processname* (blank-fill)
    `[4:7]` *#1st-qualif-name* (blank-fill)
    `[8:11]` *2nd-qualif-name* (blank-fill)

- *$devname* returns as:

  [0:11] *$devname* (blank-fill)

- *$ldevnum* returns as:

  [0:11] *$ldevnum* (blank-fill)

If any of the forms described above are preceded by "*sysname*," the result is as given above, except that "*sysnum*" replaces "$" in the result.

- *definename* returns as:

  [0:11] *definename* (blank-fill)

Any other file name is invalid.

Example

```
LENGTH := FNAMEEXPAND ( INNAME , OUTNAME , PSMG[1] );
```

FNAMETOFNAME32 Procedure (Superseded)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

FNAMETOFNAME32 converts a file name from the 12-word internal format to the 32-character Distributed Name Service (DNS) format.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
status := FNAMETOFNAME32 ( fname                  ! i
                         ,fname32 );              ! o
```

Parameters

status returned value

INT

indicates the outcome of the call.

-1 File name successfully converted
0 File name cannot be converted, or an error occurred.

fname input

STRING .EXT:ref:24

is the name to be converted. The array must contain a valid file name in 12-word internal format.

*fname*32 output

STRING .EXT:ref:32

contains, on return, the file name in DNS format. If *status* is returned as zero, the contents of this array have not been modified by the procedure.

Considerations

- If *fname* is in network format and the system number specified is not currently defined to the network, FNAMETOFNAME32 returns 0 to indicate that the file name cannot be converted.
- If a parameter is missing or a bounds error occurs on a parameter, 0 is returned.

Related Programming Manual

For network programming applications, see the *Distributed Name Service (DNS) Manual*.

FORMATCONVERT[X] Procedure

Summary

The FORMATCONVERT and FORMATCONVERTX procedures convert a format (a data record layout described by means of edit descriptors) from external form to the internal form that is required for presentation to the FORMATDATA[X] procedures. The FORMATCONVERT and FORMATCONVERTX procedures are identical, except that FORMATCONVERT requires that all of its reference parameters be 16-bit addresses, while FORMATCONVERTX accepts extended (32-bit) addresses for all of its reference parameters. For information about edit descriptors, see Appendix F, Formatter Edit Descriptors.

Syntax for C Programmers

```
#include <cextdecs(FORMATCONVERT)>

short FORMATCONVERT ( char _near *iformat
                     ,short iformatlen
                     ,char _near *eformat
                     ,short eformatlen
                     ,short _near *scales
                     ,short _near *scale-count
                     ,short conversion );


#include <cextdecs(FORMATCONVERTX)>

short FORMATCONVERTX ( char *iformat
                      ,short iformatlen
                      ,const char *eformat
                      ,short eformatlen
                      ,short  *scales
                      ,short *scale-count
                      ,short conversion );
```

Syntax for TAL Programmers

```
status := FORMATCONVERT[X] ( iformat              ! o
                            ,iformatlen           ! i
                            ,eformat              ! i
                            ,eformatlen           ! i
                            ,scales               ! o
                            ,scale-count          ! i,o
                            ,conversion );        ! i
```

Parameters

status returned value

INT

is a value indicating the outcome of FORMATCONVERT[X]:

> 0 Indicates successful conversion. The value is the number of bytes in the converted format (*iformat*).

= 0 Indicates *iformatlen* was insufficient to hold the entire converted format.

< 0 Indicates an error in the format. The value is the negated byte location in the input string at which the error was detected. The first byte of *eformat* is numbered 1.

iformat output

STRING:ref:* (Use with FORMATCONVERT)
STRING .EXT:ref:* (Use with FORMATCONVERTX)

is an array in which FORMATCONVERT[X] stores the converted format. The contents of this array must be passed to the FORMATDATA[X] procedure as an integer parameter, but FORMATCONVERT requires it to be in byte-addressable G-relative storage. Thus *iformat* must be aligned on a word boundary, or the contents of *iformat* must be moved to a word-aligned area when it is passed to FORMATDATA[X]. (The area passed to FORMATDATA need not be in byte-addressable storage.)

iformatlen input

INT:value

is the length, in bytes, of the *iformat* array. If the converted format is longer than *iformatlen*, the conversion terminates and a *status* value <= 0 returns.

eformat input

STRING:ref:* (Use with FORMATCONVERT)
STRING .EXT:ref:* (Use with FORMATCONVERTX)

is the format string in external (ASCII) form.

eformatlen input

INT:value

is the length, in bytes, of the *eformat* string.

scales output

INT:ref:* (Use with FORMATCONVERT)
INT .EXT:ref:* (Use with FORMATCONVERTX)

is an integer array. FORMATCONVERT[X] processes the format from left to right, placing the scale factor (the number of digits that appear to the right of the decimal point) specified or implied by each repeatable edit descriptor into the next available element of *scales*. This is done until the last repeatable edit descriptor is converted or the maximum specified by *scale-count* is reached, whichever occurs first.

scale-count input, output

INT:ref:* (Use with FORMATCONVERT)
INT .EXT:ref:* (Use with FORMATCONVERTX)

on call, is the number of occurrences of the *scales* array.

On return, *scale-count* contains the actual number of repeatable edit descriptors converted.

If the number of repeatable edit descriptors present is greater than the number entered here, FORMATCONVERT[X] stops storing scale factors when the *scale-count* maximum is reached, but it continues to process the remaining edit descriptors and it continues incrementing *scale-count*.

Note. The *scales* parameter information is included to provide information needed by the ENFORM product. It might not interest most users of FORMATCONVERT[X]. If so, supply a variable initialized to 0 for *scales* and *scale-count*.

---

conversion input

INT:value

Specifies the type of conversion to be done:

0 Check validity of format only. No data is stored into *iformat*. The scale information is stored in the *scales* array.

1 Produce expanded form with modifiers and decorations. This requires additional storage space, but the execution time is half that of version 2 (below). The size required is approximately 10 times *eformatlen*.

2 Produce compact conversion, ignoring modifiers and decorations. The resulting format requires little storage space, but the execution time is twice as long as version 1 (above).

Related Programming Manual

For programming information about the FORMATCONVERT[X] procedure, see the *Guardian Programmer's Guide*.

FORMATDATA[X] Procedure

Summary

Note. The FORMATDATA procedure cannot be called by native processes. Although this procedure is supported for TNS processes, it should not be used for new development. Use the FORMATDATAX procedure.

The FORMATDATA (which is superseded by FORMATDATAX) and FORMATDATAX procedures convert data item values between internal and external representations, as specified by a format (previously converted from external to internal form by FORMATCONVERT[X]) or by the list-directed conversion rules. The FORMATDATA and FORMATDATAX procedures are identical, except that FORMATDATA requires that

all of its reference parameters be 16-bit addresses, while FORMATDATAX accepts extended (32-bit) addresses for all of its reference parameters.

Syntax for C Programmers

```
#include <cextdecs(FORMATDATA)>

short FORMATDATA ( char _near *buffer
                  ,short bufferlen
                  ,short buffer-occurs
                  ,short _near *length
                  ,short _near *iformat
                  ,short _near *variable-list
                  ,short variable-list-len
                  ,short flags );


#include <cextdecs(FORMATDATAX)>

short FORMATDATAX ( char *buffer
                   ,short bufferlen
                   ,short buffer-occurs
                   ,short *length
                   ,short *iformat
                   ,short *variable-list
                   ,short variable-list-len
                   ,short flags );
```

Syntax for TAL Programmers

```
error := FORMATDATA[X] ( buffer                ! i,o
                        ,bufferlen             ! i
                        ,buffer-occurs         ! i
                        ,length                ! o
                        ,iformat               ! i
                        ,variable-list         ! i
                        ,variable-list-len     ! i
                        ,flags );              ! i
```

Parameters

error returned value

INT:value

indicates the outcome of the call. Possible values are:

0	Successful operation
267	Buffer overflow
268	No buffer
270	Format loopback

271 EDIT item mismatch
272 Invalid input character
273 Bad format
274 Numeric overflow

buffer input, output

STRING:ref:* (Use with FORMATDATA)
STRING .EXT:ref:* (Use with FORMATDATAX)

is a buffer or a series of contiguous buffers where the formatted output data is placed or where the input data is found. The length, in bytes, of *buffer* must be at least *bufferlen* * *buffer-occurs*.

bufferlen input

INT:value

is the length, in bytes, of each buffer in the *buffer* array.

buffer-occurs input

INT:value

is the number of buffers in *buffer*.

length output

INT:ref:* (Use with FORMATDATA)
INT .EXT:ref:* (Use with FORMATDATAX)

is an array that must have at least as many elements as there are buffers in the *buffer* array on output. FORMATDATA[X] stores the highest referenced character position in each buffer in the corresponding *length* element. If a buffer is not accessed, -1 is stored for that buffer and for all succeeding ones. If a buffer is skipped (for example, due to consecutive buffer advance descriptors in the format), 0 is stored.

There are no values stored in the *length* parameter during the input operation.

iformat input

INT:ref:* (Use with FORMATDATA)
INT .EXT:ref:* (Use with FORMATDATAX)

is an integer array containing the internal format, constructed by a previous call to FORMATCONVERT[X].

variable-list input

INT:ref:* (Use with FORMATDATA)
INT .EXT:ref:* (Use with FORMATDATAX)

is a 4- to 7-word entry for each array or variable. See “Considerations” for the contents and form of this array.

variable-list-len input

INT:value

is the number of *variable-list* entries passed in this call.

flags input

INT:value

Bit:

<15> Input

0 FORMATDATA[X] performs output operations.
1 FORMATDATA[X] performs input operations.

<14:5>Reserved, specify 0

<4> Null value passed

0 Each *variable-list* item is a 4-word group (FORMATDATA) or a 5-word group (FORMATDATAX).

1 Each *variable-list* item is a 5-word group (FORMATDATA) or a 7-word group (FORMATDATAX).

<3> P-Relative (*iformat* array)

0 The *iformat* array address is G-relative.
1 The *iformat* array address is P-relative.

<2> List-directed (for information about list-directed operations, see the *Guardian Programmer’s Guide*)

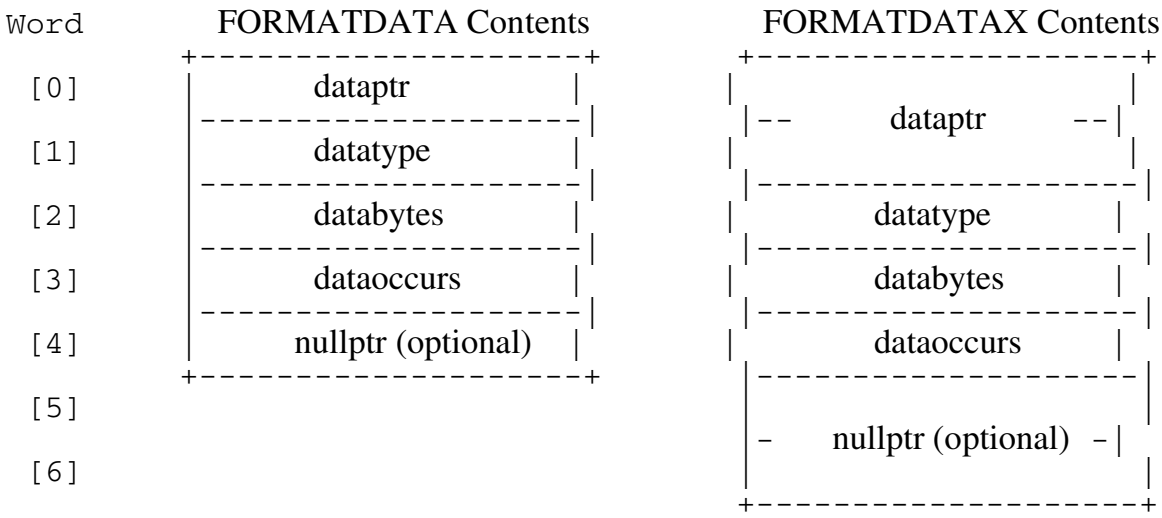
0 Apply the format-directed operation.
1 Apply the list-directed operation.

<1:0> Reserved, specify 0

Considerations

- A passed P-relative *iformat* array must be in the same code segment as the call.
- *variable-list* array form

The 4- to 7-word entry for each array or variable consists of these items:



dataptr

is the address of the array or variable. For FORMATDATA, *dataptr* is a byte address for data types 0, 1, 12-15, and 17, and is a word address for other types. For FORMATDATAX, *dataptr* is an extended address.

datatype

is the type and scale factor of the element:

bits <8:15>:

0	String
1	Numeric string unsigned
2	Integer(16) signed
3	Integer(16) unsigned
4	Integer(32) signed
5	Integer(32) unsigned
6	Integer(64) signed
7	Not used
8	Real(32)
9	Complex(32*2)
10	Real(64)
11	Complex(64*2)
12	Numeric string, sign trailing, embedded
13	Numeric string, sign trailing, separate
14	Numeric string, sign leading, embedded
15	Numeric string, sign leading, separate
16	Not used
17	Logical * 1 (1 byte)
18	Not used
19	Logical * 2 (INT(16))
20	Not used
21	Logical * 4 (INT(32))
22	Integer(8) signed
23	Integer (8) unsigned

Note. Data types 7 through 11 require floating-point firmware.

bits <0:7> Scale factor moves the position of the implied decimal point by adjusting the internal representation of the expression. Scale factor is the number of positions that the implied decimal point is moved to the left (factor > 0) or to the right (factor <= 0) of the least significant digit. This value must be 0 for data types 0, 17, 19, and 21.

databytes

is the size, in bytes, of the variable or array element used to determine the size of strings and address spacing.

dataoccurs

is the number of elements in the array *nullptr* (supply 1 for undimensioned variables).

If <> 0, it is the byte address of the null value. If = 0, there is no null value for this variable.

Example

```
ERROR := FORMATDATAX ( BUFFERS , BUF^LEN , NUM^BUFS , BUF^LENS
                     , WFORMAT , VLIST , 4 , 0 );
```

Related Programming Manual

For programming information about the FORMATDATA procedure, see the *Guardian Programmer's Guide*.

FP_IEEE_DENORM_GET_ Procedure

Summary

The FP_IEEE_DENORM_GET_ procedure reads the IEEE floating-point denormalization mode.

Note. This procedure is supported in the G06.06 RVU and all subsequent G-series RVUs. IEEE floating-point is available on all S-series processors except S70000 servers with NSR-G processors.

Syntax for C Programmers

```
#include <kfpieee.h>
fp_ieee_denorm FP_IEEE_DENORM_GET_ (void);
```

Syntax for TAL Programmers

```
?source $system.system.kfpieee
DeNorm := FP_IEEE_DENORM_GET_ ;
```

Parameters

DeNorm output

INT(32)

The denormalization control mode.

DeNorm can have these values:

FP_IEEE_DENORMALIZATION_ENABLE	Denormalization in IEEE floating point allows for greater precision in the representation of numbers that are very close to zero. This is the standard mode.
FP_IEEE_DENORMALIZATION_DISABLE	The nonstandard mode. When denormalization is disabled, fractions that are too small to be represented in standard IEEE form are represented as zero, causing a loss of precision.

FP_IEEE_DENORM_SET_ Procedure

Summary

The FP_IEEE_DENORM_SET_ procedure sets the IEEE floating-point denormalization mode.

Note. This procedure is supported in the G06.06 RVU and all subsequent G-series RVUs. IEEE floating-point is available on all S-series processors except S70000 servers with NSR-G processors.

Syntax for C Programmers

```
#include <kfpieee.h>
void FP_IEEE_DENORM_SET_( fp_ieee_denorm new_mode );
```

Syntax for TAL Programmers

```
?source $system.system.kfpieee
FP_IEEE_DENORM_SET_ ( NewMode );      ! i
```

Parameters

NewMode input

INT(32)

The denormalization control mode.

NewMode can have these values:

FP_IEEE_DENORMALIZATION_ENABLE	Denormalization in IEEE floating point allows for greater precision in the representation of numbers that are very close to zero. This is the standard mode.
FP_IEEE_DENORMALIZATION_DISABLE	The nonstandard mode. When denormalization is disabled, fractions that are too small to be represented in standard IEEE form are represented as zero, causing a loss of precision.

Consideration

Operations with denormalization disabled can cause problems by causing a gap around zero in the distribution of values that can be represented. With denormalization disabled, the results will not comply with the IEEE standard and might not match results on any other system.

FP_IEEE_ENABLES_GET_ Procedure

Summary

The FP_IEEE_ENABLES_GET_ procedure reads the IEEE floating-point trap enable mask. A set bit (value of one) means that the trap for that particular exception is enabled. A zero bit means that it is disabled.

Note. This procedure is supported in the G06.06 RVU and all subsequent G-series RVUs. IEEE floating-point is available on all S-series processors except S70000 servers with NSR-G processors.

Syntax for C Programmers

```
#include <kfpieee.h>
fp_ieee_enables FP_IEEE_ENABLES_GET_(void);
```

Syntax for TAL Programmers

```
?source $system.system.kfpieee
Traps := FP_IEEE_ENABLES_GET_ ;
```

Parameters

Traps input

INT(32)

The 32-bit trap enable mask.

Mask bit values of *Traps* are:

FP_IEEE_ENABLE_INVALID	Trap on FP_IEEE_INVALID exception.
FP_IEEE_ENABLE_DIVBYZERO	Trap on FP_IEEE_DIVBYZERO exception.
FP_IEEE_ENABLE_OVERFLOW	Trap on FP_IEEE_OVERFLOW exception.
FP_IEEE_ENABLE_UNDERFLOW	Trap on FP_IEEE_UNDERFLOW exception.
FP_IEEE_ENABLE_INEXACT	Trap on FP_IEEE_INEXACT exception.

Considerations

- A constant named FP_IEEE_ALL_ENABLES is equivalent to a combination of the mask bits to enable traps for all the exceptions.
- In some cases, the conditions that cause a trap are slightly different from the conditions that cause the corresponding exception flag to be set.
- When a trap happens, a `SIGFPE` signal is raised, and the corresponding signal handler is called. The `SIGFPE` signal handler typically does a function frame trace showing the point of failure, and then abends the process. The `SIGFPE` signal is not allowed to return to the point where the trap happened.
- Trap handling is an optional part of the IEEE floating-point standard. See FP_IEEE_EXCEPTIONS_GET_ Procedure on page 5-253 and FP_IEEE_EXCEPTIONS_SET_ Procedure on page 5-255 for an alternative to using traps.
- The compiler optimizer might reorder operations within a local routine and cause different results from the FP_ IEEE status procedures than intended. To work around this, place arithmetic operations in a separate function. The compiler cannot optimize across function boundaries, so the FP_IEEE status procedure will be called in the intended order.

FP_IEEE_ENABLES_SET_ Procedure

Summary

The FP_IEEE_ENABLES_SET_ procedure sets the IEEE floating-point trap enable mask. A set bit (value of one) enables a trap for the particular exception. A zero bit (the normal value) disables that trap.

Note. This procedure is supported in the G06.06 RVU and all subsequent G-series RVUs. IEEE floating-point is available on all S-series processors except S70000 servers with NSR-G processors.

Syntax for C Programmers

```
#include <kfpieee.h>
void FP_IEEE_ENABLES_SET_( fp_ieee_enables new_mask );
```

Syntax for TAL Programmers

```
?source $system.system.kfpieee
FP_IEEE_ENABLES_SET_ ( NewMask );      ! i
```

Parameters

NewMask input

INT(32)

The 32-bit traps flag.

Traps flag values of *Traps* are:

FP_IEEE_ENABLE_INVALID	Trap on FP_IEEE_INVALID exception.
FP_IEEE_ENABLE_DIVBYZERO	Trap on FP_IEEE_DIVBYZERO exception.
FP_IEEE_ENABLE_OVERFLOW	Trap on FP_IEEE_OVERFLOW exception.
FP_IEEE_ENABLE_UNDERFLOW	Trap on FP_IEEE_UNDERFLOW exception.
FP_IEEE_ENABLE_INEXACT	Trap on FP_IEEE_INEXACT exception.

Considerations

- When you enable traps, you will not get a trap from a left-over status; you will trap only from operations that happen after you enable the traps.
- For more considerations for this procedure, see Considerations on page 5-248.

Examples

C Example

```
#include <kfpieee.h>

void TrapsEnableExample(void) {

   FP_IEEE_ENABLES_SET_
          ( FP_IEEE_ENABLE_INVALID
            FP_IEEE_ENABLE_DIVBYZERO
            FP_IEEE_ENABLE_OVERFLOW
          );
}
```

This sets traps on the FP_IEEE_INVALID, FP_IEEE_DIVBYZERO, and FP_IEEE_OVERFLOW exceptions.

TAL Example

```
?nolist
?source $system.system.kfpieee
?list

proc TrapsEnableExample;

begin

   call FP_IEEE_ENABLES_SET_
          (   FP_IEEE_ENABLE_INVALID
         LOR FP_IEEE_ENABLE_DIVBYZERO
         LOR FP_IEEE_ENABLE_OVERFLOW
          );

end;
```

FP_IEEE_ENV_CLEAR_ Procedure

Summary

The FP_IEEE_ENV_CLEAR_ procedure sets the floating-point environment (consisting of the rounding mode, the exception flags, the trap enables, and the denormalization mode) back to its initial values. The initial values are as follows:

Rounding mode Round to nearest or nearest even value

Exception flags	No exceptions encountered (zeroes)
Trap enables	All floating-point traps disabled
Denormalization	Denormalized enabled

Note. This procedure is supported in the G06.06 RVU and all subsequent G-series RVUs. IEEE floating-point is available on all S-series processors except S70000 servers with NSR-G processors.

Syntax for C Programmers

```
#include <kfpieee.h>
fp_ieee_env FP_IEEE_ENV_CLEAR_(void);
```

Syntax for TAL Programmers

```
?source $system.system.kfpieee
SavedEnv := FP_IEEE_ENV_CLEAR_ ;
```

Parameters

SavedEnv input

INT(32)

The current floating-point environment is saved here before it is set to its initial values.

Consideration

FP_IEEE_ENV_CLEAR_ and FP_IEEE_ENV_RESUME_ are for use by a process, such as a signal handler, a clean-up routine, or a procedure that needs to tolerate being called with any possible values in the floating-point status and control. They are not for use by interrupt handlers.

Examples

C Example

```
#include <kfpieee.h>

void TotalEnvExample(void) {
   fp_ieee_env previousEnv;
   previousEnv = FP_IEEE_ENV_CLEAR_(); /*restore initial env*/
   Do_Computation();
   FP_IEEE_ENV_RESUME_( previousEnv )  /*restore previous env*/
}
```

TAL Example

```
proc DOCOMPUTATION; external;

?nolist
?source $system.system.kfpieee
?list

proc TotalEnvExample;

begin

   int(32) previousEnv;

   previousEnv := FP_IEEE_ENV_CLEAR_;   ! revert to standard env
   call DOCOMPUTATION;  ! do IEEE floating-point computation
   call FP_IEEE_ENV_RESUME_( previousEnv ) ! restore saved env
end;
```

FP_IEEE_ENV_RESUME_ Procedure

Summary

The FP_IEEE_ENV_RESUME_ procedure restores the floating-point environment (the rounding mode, the exception flags, the trap enables, and the denormalization mode) to the values it had before calling FP_IEEE_ENV_CLEAR_.

Note. This procedure is supported in the G06.06 RVU and all subsequent G-series RVUs. IEEE floating-point is available on all S-series processors except S70000 servers with NSR-G processors.

Syntax for C Programmers

```
#include <kfpieee.h>
void FP_IEEE_ENV_RESUME_( fp_ieee_env savedEnv );
```

Syntax for TAL Programmers

```
?source $system.system.kfpieee
FP_IEEE_ENV_RESUME_ ( SavedEnv );      ! i
```

Parameters

SavedEnv input

INT(32)

The previous floating-point environment that was saved by the last call to FP_IEEE_ENV_CLEAR_.

Considerations

For a description of considerations for this procedure, see Consideration on page 5-251.

Examples

For an example of the use of this procedure, see Examples on page 5-251.

FP_IEEE_EXCEPTIONS_GET_ Procedure

Summary

The FP_IEEE_EXCEPTIONS_GET_ procedure reads the IEEE floating-point exception mask.

Note. This procedure is supported in the G06.06 RVU and all subsequent G-series RVUs. IEEE floating-point is available on all S-series processors except S70000 servers with NSR-G processors.

Syntax for C Programmers

```
#include <kfpieee.h>
fp_ieee_exceptions FP_IEEE_EXCEPTIONS_GET_(void);
```

Syntax for TAL Programmers

```
?source $system.system.kfpieee
Exceptions := FP_IEEE_EXCEPTIONS_GET_ ;
```

Parameters

Exceptions input

INT(32)

The 32-bit exception flags.

Exception flag values of *Exceptions* are:

Value	Cause
FP_IEEE_INVALID	Arithmetic calculations using either positive or negative infinity as an operand, zero divided by zero, the square root of -1, the rem function with zero as a divisor (which causes divide by zero), comparisons with invalid numbers, or impossible binary-decimal conversions.
FP_IEEE_DIVBYZERO	Computing x/0, where x is finite and nonzero.
FP_IEEE_OVERFLOW	Result too large to represent as a normalized number.
FP_IEEE_UNDERFLOW	Result both inexact and too small to represent as a normalized number.
FP_IEEE_INEXACT	Result less accurate than it could have been with a larger exponent range or more fraction bits. Most commonly set when rounding off a repeating fraction such as 1.0/3.0. Also set for underflow cases and some overflow cases, but not for division by zero.

Considerations

- In addition to the above enumerated constants, a constant named FP_IEEE_ALL_EXCEPTS is equivalent to a combination of all the exception bits.
- Once exception flags are set, they stay set until explicitly reset.
- More than one exception flag can result from a single floating-point operation.

Examples

C Example

```
#include <kfpieee.h>

void Example(void) {

   FP_IEEE_EXCEPTIONS_SET_( 0 );   /* clear exceptions */
   DoComputation();   /* floating-point computation */
   if( FP_IEEE_EXCEPTIONS_GET_() &
      (FP_IEEE_INVALID|FP_IEEE_OVERFLOW|FP_IEEE_DIVBYZERO)
     )

      printf( "Trouble in computation! \n" );
}
```

TAL Example

```
proc DOCOMPUTATION; external;

?nolist
?source $system.system.kfpieee
?list

literal -- return codes for Example
   NO_PROBLEM = 0D,
   TROUBLE_IN_COMPUTATION = 1D;

int(32) proc Example;

begin

   call FP_IEEE_EXCEPTIONS_SET_( 0D );   ! Clear exception bits
   call DOCOMPUTATION;       ! Routine to do IEEE fp computation

   if( FP_IEEE_EXCEPTIONS_GET_ LAND   ! test for exceptions
       ( FP_IEEE_INVALID LOR FP_IEEE_OVERFLOW LOR
FP_IEEE_DIVBYZERO )

     ) then return( TROUBLE_IN_COMPUTATION );
   return( NO_PROBLEM  );
```

FP_IEEE_EXCEPTIONS_SET_ Procedure

Summary

The FP_IEEE_EXCEPTIONS_SET_ procedure sets the IEEE floating-point exception mask.

Note. This procedure is supported in the G06.06 RVU and all subsequent G-series RVUs. IEEE floating-point is available on all S-series processors except S70000 servers with NSR-G processors.

Syntax for C Programmers

```
#include <kfpieee.h>
void FP_IEEE_EXCEPTIONS_SET_
   ( fp_ieee_exceptions new_flags );
```

Syntax for TAL Programmers

```
?source $system.system.kfpieee
FP_IEEE_EXCEPTIONS_SET_ ( NewFlags );      ! i
```

Parameters

NewFlags input

INT(32)

The 32-bit exception flags.

Exception flag values of *NewFlags* are:

FP_IEEE_INVALID	Arithmetic calculations using either positive or negative infinity as an operand, zero divided by zero, the square root of -1, the rem function with zero as a divisor (which causes divide by zero), comparisons with invalid numbers, or impossible binary-decimal conversions.
FP_IEEE_DIVBYZERO	Computing x/0, where x is finite and nonzero.
FP_IEEE_OVERFLOW	Result too large to represent as a normalized number.
FP_IEEE_UNDERFLOW	Result both inexact and too small to represent as a normalized number.
FP_IEEE_INEXACT	Result less accurate than it could have been with a larger exponent range or more fraction bits. Most commonly set when rounding off a repeating fraction such as 1.0/3.0. Also set for underflow cases and some overflow cases, but not for division by zero.

Considerations

For a description of considerations for this procedure, see Considerations on page 5-254.

Examples

For examples of the use of this call, see Examples on page 5-254.

FP_IEEE_ROUND_GET_ Procedure

Summary

The FP_IEEE_ROUND_GET_ procedure reads the current rounding mode.

Note. This procedure is supported in the G06.06 RVU and all subsequent G-series RVUs. IEEE floating-point is available on all S-series processors except S70000 servers with NSR-G processors.

Syntax for C Programmers

```
#include <kfpieee.h>
p_ieee_round FP_IEEE_ROUND_GET_(void);
```

Syntax for TAL Programmers

```
?source $system.system.kfpieee
RoundMode := FP_IEEE_ROUND_GET_ ;
```

Parameters

RoundMode input

INT(32)

The 32-bit rounding mode code.

Rounding mode values returned by this procedure are:

FP_IEEE_ROUND_NEAREST	Round toward the representable value nearest the true result. In cases where there are two equally near values, the "even" value (the value with the least-significant bit zero) is chosen (the standard rounding mode).
FP_IEEE_ROUND_UPWARD	Round up (toward plus infinity).
FP_IEEE_ROUND_DOWNWARD	Round down (toward minus infinity).
FP_IEEE_ROUND_TOWARDZERO	Round toward zero (truncate).

FP_IEEE_ROUND_SET_ Procedure

Summary

The FP_IEEE_ROUND_SET_ procedure sets the current rounding mode.

Note. This procedure is supported in the G06.06 RVU and all subsequent G-series RVUs. IEEE floating-point is available on all S-series processors except S70000 servers with NSR-G processors.

Syntax for C Programmers

```
#include <kfpieee.h>
void FP_IEEE_ROUND_SET_( fp_ieee_round new_mode );
```

Syntax for TAL Programmers

```
?source $system.system.kfpieee
FP_IEEE_ROUND_SET_ ( NewMode );      ! i
```

Parameters

NewMode input

INT(32)

The 32-bit rounding mode code.

The rounding mode can have one of these values:

FP_IEEE_ROUND_NEAREST	Round toward the representable value nearest the true result. In cases where there are two equally near values, the "even" value (the value with the least-significant bit zero) is chosen (the standard rounding mode).
FP_IEEE_ROUND_UPWARD	Round up (toward plus infinity).
FP_IEEE_ROUND_DOWNWARD	Round down (toward minus infinity).
FP_IEEE_ROUND_TOWARDZERO	Round toward zero (truncate).

6 Guardian Procedure Calls (G)

This section contains detailed reference information for all user-accessible Guardian procedure calls beginning with the letter G. Table 6-1 on page 6-1 lists all the procedures in this section.

Table 6-1. Procedures Beginning With the Letter G

GETCPCBINFO Procedure

Summary

The GETCPCBINFO procedure provides a process with information from its own (the current) process control block (PCB).

Syntax for C Programmers

```
#include <cextdecs(GETCPCBINFO)>

void GETCPCBINFO ( short request-id
                  ,short _near *cpcb-info
                  ,short in-length
                  ,short _near *out-length
                  ,short _near *error );
```

Syntax for TAL Programmers

```
CALL GETCPCBINFO ( request-id                    ! i
                  ,cpcb-info                     ! o
                  ,in-length                     ! i
                  ,out-length                    ! o
                  ,error );                      ! o
```

Parameters

request-id input

INT:value

specifies the information to be returned. Each request ID causes a value of type INT to be returned in *cpcb-info*. The list of valid request IDs are:

0 Remote creator flag; returns 1 in *cpcb-info* if creator was remote.

1 Logged-on process state; returns 1 in *cpcb-info* if the process is currently logged on.

2 Safeguard-authenticated logon flag; returns 1 in *cpcb-info* if the process was started after successfully logging on via a terminal owned by Safeguard.

3 Safeguard-authenticated logoff state; returns 1 in *cpcb-info* if the Safeguard-authenticated logon flag is set but the process has logged off.

4 Inherited-logon flag; returns 1 in *cpcb-info* if the logon was inherited by the process.

5 Stop-on-logoff flag; returns 1 in *cpcb-info* if the process is to be stopped when it requests to be placed in the logged-off state.

6 Propagate-logon flag; returns 1 in *cpcb-info* if the process's local descendants are to be created with the inherited-logon flag set.

7 Propagate-stop-on-logoff flag; returns 1 in *cpcb-info* if the process's local descendants are to be created with the stop-on-logoff flag set.

16 Logon flags and states; returns current settings of all the logon flags and state indicators in *cpcb-info*.

The bits are defined as follows:

```
<0:8> (reserved)
<9>   Propagate stop-on-logoff
<10>  Propagate logon
<11>  Stop on logoff
<12>  Inherited logon
<13>  Safeguard-authenticated logoff state
<14>  Safeguard-authenticated logon
<15>  Logged-on state
```

cpcb-info output

INT:ref:*

is an array that returns with the information requested from the PCB.

in-length input

INT:value

specifies the length, in bytes, of the *cpcb-info* array. (This is used to prevent possible data overrun.)

out-length output

INT:ref:1

specifies the number of bytes of information returned in *cpcb-info*.

error output

INT:ref:1

returns a file-system error number indicating the outcome of the PCB information request.

Example

```
CALL GETCPCBINFO ( REQUEST^ID
                 , PCB^INFO
                 , IN^LENGTH
                 , OUT^LENGTH
                 , ERROR^REQUEST );
```

GETCRTPID Procedure (Superseded by PROCESS_GETINFOLIST_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The GETCRTPID procedure is used to obtain the 4-word CRTPID (which contains the process name or creation timestamp in words [0:2] and *cpu,pin* in word [3]) associated with a process. The term CRTPID is synonymous with process ID as used in this manual.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
CALL GETCRTPID ( cpu,pin                          ! i
               ,process-id );                     ! o
```

Parameters

cpu,pin input

INT:value

is the processor number and PIN number of the process whose CRTPID is returned (see *process-id*[3] below for the format). The PIN number is used to identify a process's process control block (or PCB) in a given processor.

process-id output

INT:ref:4

is the 4-word array where GETCRTPID returns the CRTPID (or process ID) of the process specified by *cpu,pin*. The *process-id* is returned in local form, that is,

```
[0:2]     Process name or creation timestamp
  [3].<0:3>
          Reserved
        .<4:7>
          processor number where the process is executing
        .<8:15>
          PIN assigned by the operating system to identify the process in the
          processor
```

Condition Code Settings

< (CCL) indicates that GETCRTPID failed, or that no such process exists, or that the process exists but it is terminating.

= (CCE) indicates that GETCRTPID completed successfully.

> (CCG) does not return from GETCRTPID.

Considerations

- Passing the process ID to OPEN

  The process ID returned from GETCRTPID is suitable for passing directly to the file-system OPEN procedure (if expanded to 12 words and blank-filled on the right).

- An application acquiring its own process ID

  An application that is running at a low PIN can acquire its own *process-id* by passing the results of the MYPID procedure to the GETCRTPID procedure:

  ```
  CALL GETCRTPID ( MYPID, MY^PROCESSID );
  ```

  The PID of a process is NOT shorthand for the process ID. It is a term for the *cpu,pin* for a process.

- High-PIN processes

  You cannot use GETCRTPID for high-PIN processes because a high PIN cannot fit into *cpu,pin* or *process-id*.

Example

```
CALL GETCRTPID ( PID , PROCESS^ID );
```

GETDEVNAME Procedure (Superseded by DEVICE_GETINFOBYLDEV_ Procedure (Superseded on G-series RVUs) or FILENAME_FINDNEXT_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The GETDEVNAME procedure obtains the name associated with a logical device number. GETDEVNAME returns the name of a designated logical device, if such a device exists, and if the device attributes match any optional *devtype* and *devsubtype* parameters specified. If the designated logical device does not exist or does not match optional *devtype,* and *devsubtype* parameters, the search continues for the name of the next higher (numerically) logical device which does meet these criteria.

When GETDEVNAME searches for the next higher logical device and optional parameters are supplied, it returns the name of the next higher logical device that matches all supplied *sysnum*, *devtype* and *devsubtype* parameters.

A status word is returned from GETDEVNAME that indicates whether or not the designated device exists or if a higher entry exists. By repeatedly calling GETDEVNAME and supplying successively higher logical device numbers, you can obtain the names of all system devices.

Parameters *devtype* and *devsubtype* can serve as a mask for those callers interested only in a particular type or subtype of device.
By passing either *devtype* or *devsubtype* or both, the caller can exclude all devices with other types or subtypes from the search.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
status := GETDEVNAME ( ldevnum                ! i,o
                      ,devname                ! o
                      ,[ sysnum ]             ! i
                      ,[ devtype ]            ! i
                      ,[ devsubtype ] );      ! i
```

Parameters

status returnedvalue

INT

indicates the outcome of the call. The values of *status* can be:

0 Successful; the name of the designated logical device is returned in *devname*.
1 The designated logical device does not exist. The logical device number of the next higher device is returned in *ldevnum*; the name of that device is returned in *devname*.
2 There is no logical device with *ldevnum* equal to or greater than *ldevnum* which matches the *devtype* and *devsubtype* parameters, if supplied.
4 The system specified could not be accessed.
99 Parameter error.

ldevnum input, output

INT:ref:1

is the logical device number at which an ascending search for a logical device is to begin. If any of the optional *sysnum*, *devtype*, and *devsubtype* parameters are specified, only devices that match the specified parameters satisfy the search. The range of valid logical device number input values is 0 through 65375.

You can also specify a starting value of 65535 to request a search starting with the lowest-numbered logical device in the system. This alternative is equivalent to specifying a starting value of 0, and is provided only for compatibility with previous RVUs.

On return, *ldevnum* receives the number of the first matching logical device, if one exists. The *ldevnum* remains unchanged if no such logical device exists. If *ldevnum* is out of range on input, GETDEVNAME returns a *status* value of 2 (no

logical device exists whose logical device number is greater than or equal to *ldevnum*) and *ldevnum* remains unchanged.

Note. The calling program must treat *ldevnum* as unsigned. This exception exists for compatibility with previous RVUs.

It was formerly possible to request a search starting at the lowest logical device in the system by passing any negative number in *ldevnum*. This functionality still exists, but the negative number must be either -1 or the equivalent unsigned value (65535). An arbitrary negative number is no longer accepted.

---

devname output

INT:ref:4

returns the device name or volume name of the designated device, if it exists, or the next higher (numerically) logical device if the designated device does not exist. The *devname* remains unchanged if no higher logical device exists.

sysnum input

INT:value

specifies the system (in a network) that is searched for *ldevnum*. If omitted, the local system is assumed.

devtype input

INT:value

specifies an optional device type qualifier. If specified, the device type has to match the designated device. If they do not match, the device is ignored and the search continues.

devsubtype input

INT:value

specifies an optional device subtype qualifier. If specified, the device subtype has to match the designated device. If they do not match, the device is ignored and the search continues.

Considerations

- The device name is returned in network form whenever the *sysnum* parameter is supplied (except when the local system number is specified).
- If the *sysnum* parameter is supplied, devices whose names contain seven characters are not accessible using this procedure. This is because internal-form network names are limited to six characters.
- A process name is returned as a device name if you specify a logical device number that corresponds to a destination control table (DCT) entry for a process.

- If the *devname* being returned is that of a demountable disk, and the disk has been demounted or is down, GETDEVNAME returns a status of 0, and the name returned will be one of these:
  - 4 words of blanks (`"  ","  ","  ","  "`)
  - 4 words of zero (`0,0,0,0`)
  - 1 word identifying the node number and 3 words of blanks, for example, `"\n","  ","  ","  "` (*n* is the value in SYSNUM)
  - 1 word identifying the node number and 3 words of zero for example, `"\n",0,0,0` (*n* is the value in SYSNUM)

Example

```
INT      system;        !target system
INT      ldev;          !ldev to start search
INT      name [0:3] := ["        "];
STRING   names = name;
INT      status;

  ! get next disk name
  DO
     BEGIN
     ldev := ldev + 1;
     status := getdevname ( ldev, name, system, 3 );
     END
  UNTIL (status '>' 1)
  OR (name AND name <> "   "
      AND NOT (names = "\" AND (name[1] = "   "
                                OR name[1] = 0)));
```

GETINCREMENTEDIT Procedure

Summary

The GETINCREMENTEDIT procedure returns the record number increment value for an IOEdit file.

GETINCREMENTEDIT is an IOEdit procedure and can only be used with files that have been opened by OPENEDIT or OPENEDIT_.

Syntax for C Programmers

```
#include <cextdecs(GETINCREMENTEDIT)>

__int32_t GETINCREMENTEDIT ( short filenum );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
increment := GETINCREMENTEDIT ( filenum );        ! i
```

Parameters

increment returned value

INT(32)

returns the record number increment value for the specified file, or 0 if the file is not open. The record number is 1000 times the EDIT line number.

filenum input

INT:value

specifies the file number of the open file of interest.

Related Programming Manual

For programming information about the GETINCREMENTEDIT procedure, see the *Guardian Programmer's Guide*.

GETPOOL Procedure (Superseded by POOL_* Procedures)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development. *POOL procedures are replaced by POOL_* procedures. There is no one-for-one replacement.

The GETPOOL procedure obtains a block of memory from a buffer pool.

Considerations for C Programmers

- You cannot call GETPOOL directly from a C program, because it returns a value and also sets the condition-code register. To access this procedure, you must write a "jacket" procedure in TAL that your C program can call directly. For information on how to do this, see the discussion of procedures that return a value and a condition code in the *C/C++ Programmer's Guide*. Note that the POOL_* procedures, which should be used in new development can be called directly from C.

Syntax for TAL Programmers

```
address := GETPOOL ( pool-head                ! i,o
                   ,block-size );             ! i
```

Parameters

address returned value

EXTADDR

returns the extended address of the memory block obtained if the operation is successful or returns -1D if an error occurred or *block-size* is 0. (Values less than -1D may be returned to privileged callers.)

△ **Caution.** *address* should be a simple INT(32) or EXTADDR variable; otherwise, the assignment can alter the condition code.

pool-head input,output

INT .EXT:ref:19

is the pool head previously defined by a call to DEFINEPOOL.

block-size input

INT(32):value

is the size, in bytes, of the memory obtained from the pool. This number cannot be greater than %377770D. To check data structures without getting any memory from the pool, set *block-size* to zero.

Condition Code Settings

< (CCL) indicates that *block-size* is out of range, or that the data structures are invalid; -1D is returned.

= (CCE) indicates that the operation is successful; extended address of block is returned if *block-size* is greater than zero, or -1D is returned if *block-size* is equal to 0.

> (CCG) indicates that insufficient memory is available; -1D is returned.

Considerations

- For performance reasons in the operating system, GETPOOL and PUTPOOL do not check pool data structures on each call. A process that destroys data structures or uses an incorrect address for a parameter can terminate on a call to GETPOOL or PUTPOOL: a TNS Guardian process can get an instruction failure trap (trap 1) or invalid address trap (trap 0); an OSS or native process can receive a SIGILL or SIGSEGV signal.
- In the native environment, GETPOOL verifies that all data blocks returned from the pool are aligned on a 16-byte boundary. HP suggests that code running in the TNS environment also allocate data blocks in 16-byte chunks.

Example

```
@PBLOCK := GETPOOL ( POOL^HEAD , $UDBL( $LEN( PBLOCK ) ) );
    ! get pool block size of PBLOCK in bytes.
```

Related Programming Manual

For programming information about the GETPOOL memory-management procedure, see the *Guardian Programmer's Guide*.

GETPOOL_PAGE_ Procedure (H-Series RVUs Only)

Summary

The GETPOOL_PAGE_ procedure obtains a block of memory from a buffer pool. The memory is aligned on a page boundary and the space allocated is a multiple of a page size.

Considerations for C Programmers

- You cannot call GETPOOL_PAGE_ directly from a C program, because it returns a value and also sets the condition-code register. To access this procedure, you must write a “jacket” procedure in TAL that your C program can call directly. For information on how to do this, see the discussion of procedures that return a value and a condition code in the *C/C++ Programmer’s Guide*. Note that the POOL_* procedures, which should be used in new development can be called directly from C.

Syntax for TAL Programmers

```
address := GETPOOL ( pool-head                ! i,o
                    ,block-size );            ! i
```

Parameters

address returned value

EXTADDR

returns the extended address of the memory block obtained if the operation is successful or returns -1D if an error occurred or *block-size* is 0. (Values less than -1D may be returned to privileged callers.)

△ **Caution.** *address* should be a simple INT(32) or EXTADDR variable; otherwise, the assignment can alter the condition code.

pool-head input,output

INT .EXT:ref:19

is the pool head previously defined by a call to DEFINEPOOL.

block-size input

INT(32):value

is the size, in bytes, of the memory obtained from the pool. This number cannot be greater than %377770D. To check data structures without getting any memory from the pool, set *block-size* to zero.

Condition Code Settings

< (CCL) indicates that *block-size* is out of range, or that the data structures are invalid; -1D is returned.

= (CCE) indicates that the operation is successful; extended address of block is returned if *block-size* is greater than zero, or -1D is returned if *block-size* is equal to 0.

> (CCG) indicates that insufficient memory is available; -1D is returned.

GETPOSITIONEDIT Procedure

Summary

The GETPOSITIONEDIT procedure returns the record number (1000 times the EDIT line number) of the line in the specified file most recently read or written (that is, it returns the current record number).

GETPOSITIONEDIT is an IOEdit procedure and can only be used with files that have been opened by OPENEDIT or OPENEDIT_.

Syntax for C Programmers

```
#include <cextdecs(GETPOSITIONEDIT)>

__int32_t GETPOSITIONEDIT ( short filenum );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
position := GETPOSITIONEDIT ( filenum );        ! i
```

Parameters

position returned value

INT(32)

returns the current record number (1000 times the EDIT line number) of the specified file. It returns -1 if the file is not open or if the file is positioned at its beginning. It returns -2 if the last operation was a read end-of-file.

filenum input

INT:value

specifies the file number of the open file of interest.

Related Programming Manual

For programming information about the GETPOSITIONEDIT procedure, see the *Guardian Programmer's Guide*.

GETPPDENTRY Procedure (Superseded by PROCESS_GETPAIRINFO_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The GETPPDENTRY procedure is used to obtain a description of a named process pair by its index into the destination control table (DCT). To obtain process pair descriptions by process name, use either the PROCESS_GETPAIRINFO_ procedure or the LOOKUPPROCESSNAME procedure.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
CALL GETPPDENTRY ( index                                    ! i
                  ,sysnum                                   ! i
                  ,ppd );                                   ! o
```

Parameters

index input

INT:value

specifies the index value of the DCT entry to be returned. The first entry is 0, the second is 1, and so on. The largest valid value that can be specified is 65375.

sysnum input

INT:value

specifies the system where the process pair exists.

ppd output

INT:ref:9

is an array where GETPPDENTRY returns the nine-word DCT entry specified by the given *index* and *sysnum*. Its format is:

[0:2]	Process name (in local form)
[3].<0:7>	processor of primary process
[3].<8:15>	PIN of primary process
[4].<0:7>	processor of backup process if it is a process pair. (This is 0 if there is no backup.)
[4].<8:15>	PIN of backup process, if it is a process pair. (This is 0 if there is no backup.)
[5:8]	*process-id* of ancestor. Note that the *process-id* is a 4-word array that contains:

[0:2]	Process name or creation timestamp
[3].<0:3>	Reserved
[3].<4:7>	processor number where the process is executing
[3].<8:15>	PIN assigned by the operating system to identify the process in the processor

Condition Code Settings

< (CCL) indicates that the DCT in the given system cannot be accessed.

= (CCE) indicates that the GETPPDENTRY completed successfully.

> (CCG) indicates that the *index* is greater than the last entry in the DCT.

Considerations

- Checking the DCT entry

  If *index* is not currently being used, GETPPDENTRY returns CCE and sets *ppd* to zeros. To check for all conditions, an application could contain this code:

```
CALL GETPPDENTRY( INDEX^NUM , SYS^NUM , PROCESS^PAIR^DESCRIPT
);
IF < THEN ... ; ! system unavailable.
IF > THEN ! STOP, no more DCT entries available.
IF = AND PROCESS^PAIR^DESCRIPT THEN ... ! found an entry.
ELSE
  ! unused entry, try the next INDEX^NUM.
```

- Difference between GETPPDENTRY and LOOKUPPROCESSNAME

  The difference between the GETPPDENTRY procedure and the LOOKUPPROCESSNAME procedure is:

  GETPPDENTRY is primarily used to obtain a local or remote process pair description by its index into a system table.

  LOOKUPPROCESSNAME is primarily used to obtain a local or remote process pair description by its name.

- High-PIN considerations

  If you call GETPPDENTRY for a named process pair that has a high-PIN process as the primary or backup, the *ppd* array (*ppd*[0:8]) is returned filled with zeros.

  If you call GETPPDENTRY for a named process pair that has a high-PIN process as the ancestor, a synthetic process ID is returned in *ppd*[5:8]. A synthetic process ID contains a PIN value of 255 in place of a high-PIN value, which cannot be represented by 8 bits.

GETREMOTECRTPID Procedure (Superseded by PROCESS_GETINFOLIST_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The GETREMOTECRTPID procedure is used to obtain the 4-word process ID associated with a remote process. The process ID contains the remote process name or creation timestamp in words[0:2] and *cpu,pin* in word[3]. The term CRTPID is synonymous with process ID as used in this manual.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
CALL GETREMOTECRTPID ( cpu,pin                    ! i
                      ,process-id                 ! o
                      ,sysnum );                  ! i
```

Parameters

cpu,pin input

INT:value

is the processor number and PIN number of the process whose process ID is returned (see *process-id*[3] below for the format). The PIN number is used to identify a process's process control block (or PCB) in a given processor. Note that without a system number, *cpu,pin* is not sufficient to identify a remote process in a network.

process-id output

INT:ref:4

is the 4-word array where GETREMOTECRTPID returns the process ID of the process specified by *cpu,pin*. The *process-id* is returned as follows:

[0:2]	Process name or creation timestamp
[3].<0:3>	Reserved
.<4:7>	processor number where the process is executing
.<8:15>	PIN assigned by the operating system to identify the process in the processor

If *sysnum* specifies a remote system, the process ID is in network form; if *sysnum* specifies the local system, it is in local form. The two forms differ only in the form of the process name.

A local process name consists of six bytes with the first byte being a “$” and the second containing an alphabetic character. The remaining four characters (optional) can be alphanumeric. Note that a full six character local process name cannot be converted to a remote form.

A remote process name consists of six bytes with the first byte containing a ”\” and the second containing the network system number where the process resides. The third must be an alphabetic character. The remaining three characters can be alphanumeric.

sysnum input

INT:value

is a value specifying the system from which the process ID is to be returned.

Condition Code Settings

< (CCL) indicates the GETREMOTECRTPID failed for one of these reasons:

- No such process exists.
- The process exists but it is terminating.
- The remote system could not be accessed.
- The process has an inaccessible name, consisting of more than four characters.

= (CCE) indicates that GETREMOTECRTPID was successful.

> (CCG) does not return from GETREMOTECRTPID.

Considerations

You cannot use GETREMOTECRTPID for high-PIN processes because a high PIN cannot fit into *cpu,pin* or *process-id*.

Example

```
CALL GETREMOTECRTPID ( PID , CRT^PID , SYS^NUM );
```

GETSYNCINFO Procedure (Superseded by FILE_GETSYNCINFO_ Procedure)

Summary

The GETSYNCINFO procedure is called by the primary process of a process pair before starting a series of write operations to a file open with paired access. GETSYNCINFO returns a file's synchronization block so that it can be sent to the backup process in a checkpoint message.

Note. Typically, GETSYNCINFO is not called directly by application programs. Instead, it is called indirectly by CHECKPOINT.

Syntax for C Programmers

```
#include <cextdecs(GETSYNCINFO)>

_cc_status GETSYNCINFO ( short filenum
                        ,[ short _near *sync-block ]
                        ,[ short _near *sync-block-size ] );
```

- The function value returned by GETSYNCINFO, which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL GETSYNCINFO ( filenum                          ! i
                 ,[ sync-block ]                    ! o
                 ,[ sync-block-size ] );            ! o
```

Parameters

filenum input

INT:value

is the number of an open file that identifies the file whose sync block is obtained.

sync-block output

INT:ref:*

returns the synchronization block for this file. The size, in words, of *sync-block* is determined as follows:

- For unstructured disk files, size = 8 words.
- For ENSCRIBE structured files, size in words = 11 + (longest alt key len + pri key len + 1) / 2.
- For the Transaction Management Facility, the transaction pseudofile size = 9 words.
- For processes, size = 2 words.
- For other files, size = 1 word.

sync-block-size output

INT:ref:1

returns the size, in words, of the sync block data.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILE_GETINFO_ or FILEINFO).

= (CCE) indicates that GETSYNCINFO was successful.

> (CCG) indicates that the file is not a disk file.

Considerations

- The GETSYNCINFO procedure cannot be used with Enscribe format 2 files or OSS files larger than approximately 2 gigabytes. If an attempt is made to use the GETSYNCINFO procedure with these files, error 581 is returned. For information

on how to perform the equivalent task with large files, see the FILE_GETSYNCINFO_ Procedure.

- File number has not been opened

  If the GETSYNCINFO file number does not match the file number of the open file that you are trying to access, then the call to GETSYNCINFO returns with file-system error 16.

- Buffer address out of bounds

  If an out-of-bounds application buffer address parameter is specified in the GETSYNCINFO call (that is, a pointer to the buffer has an address that is outside of the data area of the process) or if the buffer lies within the data area that is used by GETSYNCINFO, then the call returns with file-system error 22.

Example

```
CALL GETSYNCINFO ( FILE^NUM , SYNC^ID );
```

GETSYSTEMNAME Procedure (Superseded by NODENUMBER_TO_NODENAME_ Procedure)

Summary

The GETSYSTEMNAME procedure supplies the system name associated with a system number.

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
ldev := GETSYSTEMNAME ( sysnum                ! i
                       ,sysname );            ! o
```

Parameters

ldev returned value

INT

returns one of these values:

1:32766 The logical device number of the network line handler that controls the current path to the system designated by *sysnum*. The logical device number has at most 15 bits of magnitude and the specified system is accessible.

32767 Indicates one of these:

The line handler exists and the specified system is accessible, but the line handler logical device number exceeds 15 bits of magnitude.

or

The specified system is the local system, so there is no line handler logical device number to return.

In either case, the system name is returned in *sysname*.

0 The specified system does not exist.

-1 All paths to the specified system are down.

-3 Bounds error occurred on *sysname*.

sysnum input

INT:value

{0:254} is the number of the system; the name is returned in *sysname*.

sysname output

INT:ref:4

returns the name of the system corresponding to *sysnum*.

Considerations

When retrieving a line handler logical device number that exceeds 15 bits of magnitude:

GETSYSTEMNAME uses the number 32767 to represent any logical device number whose value exceeds 15 bits of magnitude. (The value 32767 is reserved and is never used as an actual logical device number.) To retrieve logical device numbers having more than 15 bits of magnitude, replace calls to GETSYSTEMNAME with calls to NODENUMBER_TO_NODENAME_.

Example

```
LDEV := GETSYSTEMNAME( SYS^NUM ,SYS^NAME );
```

GETSYSTEMSERIALNUMBER Procedure

Summary

The GETSYSTEMSERIALNUMBER procedure returns the system serial number of the caller's system as an ASCII character string of numerals.

Syntax for C Programmers

```
#include <cextdecs(GETSYSTEMSERIALNUMBER)>

 short GETSYSTEMSERIALNUMBER ( short *string-buffer
                             , short max-buffer-length
                             , short *string-length );
```

Syntax for TAL Programmers

```
error := GETSYSTEMSERIALNUMBER ( <string-buffer >          ! o
                               , <max-buffer-length >      ! i
                                 , <string-length >      );!o
```

Parameters

error returned value

INT

returns -1 if *max-buffer-length* is too small. Otherwise, it returns a file-system error value.

string-buffer output

INT .EXT:ref:*

is the string array that contains the numerals of the system serial number on return.

max-buffer-length input

INT:value

is the size of the string array buffer *string-buffer*.

string-length output

INT .EXT:ref:1

returns the number of numerals in the serial number that is returned in *string-buffer*.

Example

This example calls GETSYSTEMSERIALNUMBER and displays the result.

```
main()
{
#define MAX_ID_LEN 60

        char  idbuf[MAX_ID_LEN];

        short error;
        short idlen;
        if (error = GETSYSTEMSERIALNUMBER(idbuf, MAX_ID_LEN,
             &idlen))
                printf("GETSYSTEMSERIALNUMBER error %d\n",
error);
       else {
                idbuf[idlen] = '\0';
                printf("System serial number is %s\n", idbuf);
        }
}
```

GIVE^BREAK Procedure

Summary

The GIVE^BREAK procedure returns BREAK to the previous owner (the process that owned BREAK before the last call to TAKE^BREAK).

GIVE^BREAK is a sequential I/O (SIO) procedure and should be used only with files that have been opened by OPEN^FILE.

Syntax for C Programmers

```
#include <cextdecs(GIVE_BREAK)>

short GIVE_BREAK ( short { _near *common-fcb }
                         { _near *file-fcb   } );
```

Syntax for TAL Programmers

```
error := GIVE^BREAK ( { common-fcb }          ! i
                      { file-fcb   } );       ! i
```

Parameters

error returned value

INT

returns a file-system or sequential I/O procedure error indicating the outcome of the operation.

common-fcb input

INT:ref:*

identifies the file returning BREAK to the previous owner. The *common-fcb* parameter is allowed for convenience. If BREAK is not owned, this call is ignored.

file-fcb input

INT:ref:*

identifies the file returning BREAK to the previous owner. If BREAK is not owned, this call is ignored.

Example

```
CALL GIVE^BREAK ( OUT^FILE );          ! return BREAK to
                                       ! previous owner.
```

Related Programming Manual

For programming information about the GIVE^BREAK procedure, see the *Guardian Programmer's Guide*.

GROUP_GETINFO_ Procedure

Summary

The GROUP_GETINFO_ procedure returns attributes of the specified group, such as the group's textual description and whether the group is automatically deleted when the last member is deleted. The group can be identified by group name or group ID.

Syntax for C Programmers

Note. In the TNS/E environment the CEXTDECS file uses the *int* data type for 32-bit values. This is a change from the TNS and TNS/R environments where CEXTDECS uses the *long* data type for 32-bit values.

```
#include <cextdecs(GROUP_GETINFO_)>

short GROUP_GETINFO_ ( [ char *group-name ]
                      ,[ short group-maxlen ]
                      ,[ short *group-curlen ]
                      ,[ __int32_t *groupid ]
                      ,[ short *is-auto-delete ]
                      ,[ char *descrip ]
                      ,[ short descrip-maxlen ]
                      ,[ short *descriplen ] );
```

Syntax for TAL Programmers

```
error := GROUP_GETINFO_ ( [ group-name:group-maxlen ]   !
i,o:i
                         ,[ group-curlen ]              ! i,o
                         ,[ groupid ]                   ! i,o
                         ,[ is-auto-delete ]            ! o
                         ,[ descrip:descrip-maxlen ]    ! o:i
                         ,[ descriplen ] );             ! o
```

Parameters

error returned value

INT

returns a file-system error number indicating the outcome of the call. Common errors returned are:

0	No error.
11	Record not in file. The specified group name or group ID is undefined.
22	Parameter out of bounds. An input parameter is not within the valid range, or return information does not fit into the length of the space provided, or an output parameter overlays the stack marker that was created by calling this procedure.
29	Missing parameter. This procedure was called without specifying a required parameter.
590	Bad parameter value. Either the value specified in *group-curlen* is greater than the value specified in *group-maxlen*, the value specified in

group-curlen is not within the valid range, or the value specified in *group-id* is not within the valid range.

For more information on file-system error messages, see the *Guardian Procedure Errors and Messages Manual*.

group-name:group-maxlen input,output:input

STRING .EXT:ref:*, INT:value

on input, if present and if *group-curlen* is not 0, *group-name* specifies the group name for which information is to be returned.

On output, if *groupid* is specified and *group-curlen* is set to 0, returns the group name corresponding to the group ID specified.

group-name is passed, and returned, in the form of a case-sensitive string that is up to 32 alphanumeric characters long.

group-maxlen specifies the length of the string variable *group-name* in bytes.

This parameter pair is required if *group-curlen* is specified.

group-curlen input,output

INT .EXT:ref:1

on input, if *group-name* is specified, contains the actual length of *group-name* in bytes. The default value is 0.

On output, if *group-name* is returned, this parameter contains the actual length of *group-name* in bytes.

This parameter is required if *group-name:group-maxlen* is specified.

groupid input,output

INT(32) .EXT:ref:1

on input, if *group-curlen* is 0 or omitted, specifies the group ID for which information is to be returned.

On output, if *group-name* is specified and *group-curlen* is not 0, this parameter returns the group ID corresponding to the specified group name.

group-id is a value in the range 0 through 65,535.

is-auto-delete output

INT .EXT:ref:1

indicates whether the specified group is automatically deleted when it no longer contains members. This parameter returns these values:

-1 The group is deleted when it becomes empty.

0 The group is not deleted when it becomes empty.

descrip:descrip-maxlen ouput:input

STRING .EXT:ref:*, INT:value

if present and if *descrip-maxlen* is not 0, returns a string containing a description of the specified group. The maximum length of *descrip* is 255.

descrip-maxlen specifies the length of the string variable *descrip* in bytes.

This parameter pair is required if *descriplen* is specified.

descriplen output

INT .EXT:ref:1

is the length in bytes of the string returned in *descrip*.

This parameter is required if *descrip:descrip-maxlen* is specified.

Considerations

Either *group-name* or *group-id* must be supplied. If both parameters are supplied and *group-curlen* is greater than zero, *group-name* is treated as an input parameter and *group-id* is treated as an output parameter. If both parameters are supplied and *group-curlen* is zero, *group-id* is treated as an input parameter.

Example

```
!get the descriptive text, if any, on the specified group ID
error :=
   GROUP_GETINFO_ ( ,,group^id,,descrip:maxlen,real^len );
```

GROUP_GETNEXT_ Procedure

Summary

The GROUP_GETNEXT_ procedure returns a group name and group ID. On successive calls, all group names and group IDs can be obtained.

Syntax for C Programmers

Note. In the TNS/E environment the CEXTDECS file uses the *int* data type for 32-bit values. This is a change from the TNS and TNS/R environments where CEXTDECS uses the *long* data type for 32-bit values.

```
#include <cextdecs(GROUP_GETNEXT_)>

short GROUP_GETNEXT_ ( [ char *group-name ]
                      ,[ short group-maxlen ]
                      ,[ short *group-curlen ]
                      ,[ __int32_t *groupid ] );
```

Syntax for TAL Programmers

```
error := GROUP_GETINFO_ ( group-name:group-maxlen   ! i,o:i
                         ,group-curlen              ! i,o
                         ,[ groupid ]  );           ! o
```

Parameters

error returned value

INT

returns a file-system error number indicating the outcome of the call. Common errors returned are:

0	No error.
11	Record not in file. There are no more groups, or the specified group name is undefined.

22 Parameter out of bounds. An input parameter is not within the valid range, or return information does not fit into the length of the space provided, or an output parameter overlays the stack marker that was created by calling this procedure.

29 Missing parameter. This procedure was called without specifying a required parameter.

590 Bad parameter value. The value specified in *group-curlen* is greater than the value specified in *group-maxlen*.

For more information on file-system error messages, see the *Guardian Procedure Errors and Messages Manual*.

group-name:group-maxlen input,output:input

STRING .EXT:ref:*, INT:value

on input, if *group-curlen* is not 0, *group-name* specifies a character string that precedes the next *group-name* to be returned. *group-maxlen* specifies the length of the string variable *group-name* in bytes. To obtain the first group name, set *group-curlen* to 0.

On output, this parameter returns the group name that follows the *group-name* specified as the input parameter.

The *group-name* parameter is passed, and returned, in the form of a case-sensitive string that is as many as 32 alphanumeric characters long.

group-curlen input,output

INT .EXT:ref:1

on input, contains the actual length of *group-name* in bytes. To obtain the first group name, set *group-curlen* to 0. The default value is 0.

On output, this parameter contains the actual length of *group-name* in bytes.

groupid output

INT(32) .EXT:ref:1

returns the group ID corresponding to the returned *group-name*.

groupid is a value in the range 0 through 65535.

Considerations

- Names are not returned in any particular order, and the order can change from one RVU to the next.
- Naming rules in the Guardian environment are more restrictive than those in the OSS environment.

Example

```
!  obtain all group names
i := 0;
curlen := 0;
DO
   error := GROUP_GETNEXT_ ( name:MAXLEN, curlen);
   group^list[i] ':=' name for curlen BYTES;
   i := i + 1;
UNTIL (error <> 0);
```

GROUPIDTOGROUPNAME Procedure (Superseded by GROUP_GETINFO_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development. This procedure does not support file-sharing groups.

The GROUPIDTOGROUPNAME procedure returns the group name associated with an existing group ID from the USERID file.

Syntax for C Programmers

```
#include <cextdecs(GROUPIDTOGROUPNAME)>

_cc_status GROUPIDTOGROUPNAME ( short _near *id-name );
```

- The function value returned by GROUPIDTOGROUPNAME, which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL GROUPIDTOGROUPNAME  ( id-name ) ;              ! i,o
```

Parameters

id-name input, output

INT:ref:4

On input, contains the group ID to be converted to a group name. The group ID is passed in the form:

id-name.<8:15> = group ID {0:255}

On return, contains the group name associated with the specified group ID in the form:

id-name FOR 4 = group name (blank-filled)

Condition Code Settings

< (CCL) indicates that a required parameter is missing, that a buffer is out of bounds, or that an I/O error occurred when accessing the $SYSTEM.SYSTEM.USERID file.

= (CCE) indicates that the designated group name is returned.

> (CCG) indicates that the specified group ID is undefined.

Example

```
INT   .NAME^ID [ 0:3 ] := 8;   !this is ACCTING group!
  .
  .
  .
CALL GROUPIDTOGROUPNAME (NAME^ID);  !on return,
                                    !name^id = "ACCTING "
```

GROUPMEMBER_GETNEXT_ Procedure

Summary

The GROUPMEMBER_GETNEXT_ procedure returns a user member or alias associated with a group ID. On successive calls, all user members and aliases associated with a given group ID can be obtained.

Syntax for C Programmers

```
#include <cextdecs(GROUPMEMBER_GETNEXT_)>

short GROUPMEMBER_GETNEXT_ ( __int32_t groupid
                            ,char *member-name
                            ,short member-maxlen
                            ,short *member-curlen );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
error := GROUPMEMBER_GETNEXT_
             ( groupid                                  ! i
              ,member-name:member-maxlen                ! i,o:i
              ,member-curlen );                         ! i,o
```

Parameters

error returned value

INT

returns a file-system error number indicating the outcome of the call. Common errors returned are:

0	No error.
11	Record not in file. The specified group has no more members or is undefined.
22	Parameter out of bounds. An input parameter is not within the valid range, or return information does not fit into the length of the space provided, or an output parameter overlays the stack marker that was created by calling this procedure.
29	Missing parameter. This procedure was called without specifying all parameters.
590	Bad parameter value. Either the value specified in *member-curlen* is greater than the value specified in *member-maxlen*, the value specified in *member-curlen* is not within the valid range, or the value specified in *group-id* is not within the valid range.

For more information on file-system error messages, see the *Guardian Procedure Errors and Messages Manual*.

groupid input

INT(32):value

specifies the group ID whose associated user member or alias is to be returned. The group ID is a value in the range 0 through 65535.

member-name:member-maxlen input, output:input

STRING .EXT:ref:*, INT:value

on input, specifies a character string that precedes the next *member-name* to be returned. *member-maxlen* specifies the length of the string variable *member-name* in bytes. To obtain the first user member or alias, set *member-curlen* to 0.

On output, this parameter returns the user member or alias that follows the *member-name* specified as the input parameter.

member-name is passed, and returned, in one of two forms:

group name.user member

The group name and user member are each up to 8 alphanumeric characters long, and the first character must be a letter. The group name and user member are separated by a period (.).

alias

The alias is a case-sensitive string made up of 1 to 32 alphanumeric characters, periods (.), hyphens (-), or underscores (_). The first character must be alphanumeric.

member-curlen input, output

INT .EXT:ref:1

on input, if *member-name* is specified, contains the actual length of *member-name* in bytes. To obtain the first name, set *member-curlen* to 0.

On output, this parameter returns the actual length of *member-name* in bytes.

Considerations

- Aliases are defined only when Safeguard is installed.
- Names are not returned in any particular order, and the order can change from one RVU to the next.
- Naming rules in the Guardian environment are more restrictive than those in the OSS environment.

Example

```
!  obtain all names associated with a particular group ID
i := 0;
user^list.len[i] := 0;
DO
error := GROUPMEMBER_GETNEXT_
    (group^id, user^list.name[i]: maxlen, user^list.len[i]);
user^list.len [i + 1] := user^list.len[i];
user^list.name[i + 1] ':='
    user^list.name[i] FOR user^list.len[i] BYTES;
i := i + 1;
UNTIL (error <> 0);
```

GROUPNAMETOGROUPID Procedure (Superseded by GROUP_GETINFO_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development. This procedure does not support file-sharing groups.

The GROUPNAMETOGROUPID procedure returns the group ID associated with an existing group name from the USERID file.

Syntax for C Programmers

```
#include <cextdecs(GROUPNAMETOGROUPID)>

_cc_status GROUPNAMETOGROUPID ( short _near *name-id );
```

- The function value returned by GROUPNAMETOGROUPID, which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL GROUPNAMETOGROUPID ( name-id );          ! i,o
```

Parameters

name-id input, output

INT:ref:4

on input, *name-id* contains the group name to be converted (in place) to a group ID. The group name is passed in the form:

name-id FOR 4 = group name (blank filled)

The group name must be input in uppercase.

On return, *name-id* contains the group ID associated with the group name in the form:

name-id.<8:15> = group ID {0:255}

Condition Code Settings

< (CCL) indicates that *name-id* is out of bounds, or that an I/O error occurred when the procedure accessed the $SYSTEM.SYSTEM.USERID file.

= (CCE) indicates that the designated group ID is returned.

> (CCG) indicates that the specified group name is undefined.

Example

```
INT .NAMEID [0:3] := ["GATORS  "]; !want to get group ID
      .
      .
CALL GROUPNAMETOGROUPID (NAMEID);
!on return, NAMEID[0].<8:15> = contains the group ID
IF <> THEN ...                      !error occurred
```

7

Guardian Procedure Calls (H-K)

This section contains detailed reference information for all user-accessible Guardian procedure calls beginning with the letters H through K. Table 7-1 lists all the procedures in this section.

Table 7-1. Procedures Beginning With the Letters H Through K

HALTPOLL Procedure

Summary

The HALTPOLL procedure is normally used to stop continuous polling.

Syntax for C Programmers

```
#include <cextdecs(HALTPOLL)>

_cc_status HALTPOLL ( short filenum );
```

- The function value returned by HALTPOLL, which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL HALTPOLL ( filenum );                    ! i
```

Parameters

filenum input

INT:value

is the number of an open file that HALTPOLL stops polling.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILE_GETINFO_ or FILEINFO).

= (CCE) indicates that the HALTPOLL procedure executed successfully.

> (CCG) indicates that an error occurred (call FILE_GETINFO_ or FILEINFO).

Example

```
CALL HALTPOLL ( FNUM );
```

FNUM is the integer returned from the call to FILE_OPEN_ or OPEN that opened the particular communication line. HALTPOLL forces the immediate termination of an outstanding nowait read operation within a point-to-point station, or it stops any polling that is in progress within a multipoint station.

Related Programming Manuals

For programming information about the HALTPOLL procedure, see the data communication manuals.

HEADROOM_ENSURE_ Procedure

Summary

Note. This procedure can be called only from a native process. Its pTAL syntax is declared only in the EXTDECS0 file.

The HEADROOM_ENSURE_ procedure allows you to check that the current stack has enough room for the needs of your process. The default value of 0D indicates that the main stack can grow to 1 MB in the TNS/R enviroment and to 2 MB in the TNS/E environment. Note there are two stacks in the TNS/E environment: the memory stack and the RSE backing store. For most processes, the default value is adequate. This procedure can help you, for example, when specifying parameters for the PROCESS_LAUNCH_ procedure.

Syntax for C Programmers

```
#include <cextdecs(HEADROOM_ENSURE_)>

__int32_t HEADROOM_ENSURE_ ( __int32_t room );
```

- CEXTDECS (through the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
ret-val := HEADROOM_ENSURE_ ( room );        ! i
```

Parameters

ret-val returned value

INT(32)

returns the number of bytes between the current stack pointer and the stack limit if *room* is <= 0D. If *room* is > 0D, *ret-val* indicates the outcome of the operation. It returns one of these values:

0D	Either the requested space already exists in the stack space or the stack space was successfully enlarged to make enough room for the request.
5D	The request would have exceeded the maximum stack size.
6D	The stack pointer is invalid.
7D	The stack pointer does not address a main stack or a privileged stack.
36D	The system was unable to allocate memory.
43D	The system was unable to obtain swap space.
45D	The Kernel Managed-Swap Facility (KMSF) was unable to obtain swap space.

room input

INT(32):value

in the TNS/R environment, specifies the additional space in bytes to be allocated to the stack if *room* is > 0D. If *room* is <= 0D, HEADROOM_ENSURE_ returns the current headroom; that is, the number of bytes between the current stack pointer and the limit of the stack. In the TNS/E environment, specifies the minimum size of either the memory stack or the RSE backing store. When *room* is <=0D, it is the size of the remaining memory stack (calculated as the difference between the maximum stack and the used stack amount.)

Considerations

- If HEADROOM_ENSURE_ returns an error (not 0D), the stack is not enlarged.
- If HEADROOM_ENSURE_ is called from a user (nonprivileged) process, the main stack of the process is the target. If the call is made from a privileged procedure, the privileged stack is the target.
- The amount by which the stack is enlarged might be greater than the value specified in *room*. The stack size is rounded up by a unit determined by the

system.

Example

```
error := HEADROOM_ENSURE_ ( room );
```

HEAPSORT Procedure

Summary

The HEAPSORT procedure is used to sort an array of equal-sized elements in place.

Syntax for C Programmers

```
#include <cextdecs(HEAPSORT)>

short HEAPSORT ( short _near *array
                ,short num-elements
                ,short size-of-element
                ,short (*)()compare-function );
```

- The *compare-function* parameter is an application-supplied comparison function that must be written in C. It must return values as described earlier in this subsection under the *compare-proc* parameter in the TAL syntax.

Syntax for TAL Programmers

```
CALL HEAPSORT ( array                              ! i,o
               ,num-elements                       ! i
               ,size-of-element                    ! i
               ,compare-proc );                    ! i
```

Parameters

array input, output

INT:ref:*

contains equal-sized elements to be sorted.

num-elements input

INT:value

is the number of elements in *array*.

size-of-element input

INT:value

is the size, in words, of each element in *array*.

compare-proc input

INT PROC

is an application-supplied function procedure that HEAPSORT calls to determine the sorted order (ascending or descending) of the elements in *array*.

This procedure must be of the form:

```
INT PROC compare-proc ( element-a , element-b );
       INT .element-a;
       INT .element-b;
```

The *compare-proc* must compare *element-a* with *element-b* and return either of these values:

0 (indicating false) if *element-b* should precede *element-a*
1 (indicating true) if *element-a* should precede *element-b*

element-a and *element-b* are INT:ref parameters.

Considerations

- In addition to its local variables, HEAPSORT allocates stack space equal to the value you specify as the size of one array element. If insufficient stack space is available, the call to HEAPSORT fails: a TNS Guardian process gets a stack overflow trap; an OSS or native process receives a SIGSTK signal.

Example

In this example, HEAPSORT sorts the elements in `array` in ascending order.

```
CALL HEAPSORT ( array, num^elements, element^size,
                ascending );
```

HEAPSORTX_ Procedure

Summary

The HEAPSORTX_ procedure is used to sort an array of equal-sized elements in place.

Syntax for C Programmers

Note. In the TNS/E environment, the CEXTDECS file uses the *int* data type for 32-bit values. This is a change from the TNS and TNS/R environments where CEXTDECS uses the *long* data type for 32-bit values.

```
#include <cextdecs(HEAPSORTX_)>

short HEAPSORTX_ ( short *array
                  ,__int32_t num-elements
                  ,short size-of-element
                  ,short (*)()compare-function
                  ,__int32_t _far *pointer-array );
```

- *compare-function* is an application-supplied comparison function that must be written in C. It must return values as described under the *compare-proc* parameter in the TAL syntax, earlier in this subsection.

Syntax for TAL Programmers

```
error := HEAPSORTX_ ( array                          ! i,o
                     ,num-elements                   ! i
                     ,size-of-element                ! i
                     ,compare-proc                   ! i
                     ,[ pointer-array ] );           ! i
```

Parameters

error returned value

INT

indicates the outcome of the operation. It returns one of these values:

0	Array has been successfully sorted.
29	Required parameter is missing.
590	Invalid parameter supplied.
632	Insufficient stack space for temporary variable (see “Considerations”).

array input, output

INT .EXT:ref:*

contains equal-sized elements to be sorted.

num-elements input

INT(32):value

is the number of elements in *array*.

size-of-element input

INT:value

is the size, in words, of each element in *array*.

compare-proc input

INT PROC

is an application-supplied function procedure that HEAPSORTX_ calls to compare two array elements and determine their sorted order. The addresses of these elements are supplied as parameters to *compare-proc*.

This procedure must be of the form:

```
INT PROC compare-proc ( element-a , element-b );
       INT .EXT  element-a;
       INT .EXT  element-b;
```

The *compare-proc* must compare *element-a* with *element-b* and return a result. It must return zero (false) if *element-b* should precede *element-a*; it must return a nonzero value (true) if *element-a* should precede *element-b*.

pointer-array input

INT(32) .EXT:ref:*

provides space for an array that is used to optimize the sort. The size of each element in the array is an INT(32) and the number of elements is equal to the value specified for the *num-elements* parameter. You do not need to supply any data in the array, nor should you expect any useful data in the array on return. The value that you supply is simply a pointer to an area that has been allocated by your program that is of sufficient size for the array. The array is used as a work area by the sort.

If this parameter is specified, and not equal to 0D, the sort builds an array of pointers in the area supplied by this parameter. It is these pointers that are rearranged as the sort progresses. Only when the sort is complete is the actual data supplied in the *array* parameter rearranged. If this parameter is omitted or specified as 0D, the sort works directly on the data supplied in the *array* parameter. Supplying this parameter can substantially improve the performance of the sort, especially if there is a large number of elements or a large element size.

Considerations

- In addition to its local variables, HEAPSORTX_ allocates stack space equal to the value you specify as the size of one array element. If insufficient stack space is available, HEAPSORTX_ returns an error 632.

Example

In this example, HEAPSORTX_ sorts the elements in ARRAY in ascending order.

```
CALL HEAPSORTX_ ( array, num^elements, element^size,
                  ascending );
```

Related Programming Manual

For programming information about the HEAPSORTX_ procedure, see the *Guardian Programmer's Guide*.

HIST_FORMAT_ Procedure

Summary

The HIST_FORMAT_ procedure produces an ASCII text representation of the process state whose context is established by a previous call to the HIST_INIT_ procedure or HIST_GETPRIOR_ procedure. See the HIST_INIT_ Procedure for an overview of how HIST_INIT_, HIST_FORMAT_, and HIST_GETPRIOR_ can be used together to perform stack tracing. This procedure displays RISC register contents for TNS/R code and IPF register contents TNS/E code.

Syntax for C Programmers

```
#include <histry.h>

short HIST_FORMAT_ ( NSK_histWorkspace *workspace
                    ,char *text
                    ,const uint16 limit );
```

Syntax for TAL Programmers

```
?SOURCE $SYSTEM.SYSTEM.HHISTRY

ret-val := HIST_FORMAT_ ( workspace            !i,o
                         ,text                 !o
                         ,limit );             !i
```

Parameters

ret-val returned value

INT

if the procedure is successful, returns the length of the output text in bytes or zero (0) if there is no more text for the current context. If the procedure is unsuccessful, *ret-val* contains a negative value as follows:

-2 HIST_BAD_WIDTH

The value of the *limit* parameter is less than the minimum width for output defined by the HIST_MinWidth literal in the HHISTRY header file.

-9 HIST_BAD_WORKSPACE

The workspace structure has an invalid version identifier. This error can occur if HIST_FORMAT_ is called without first calling the HIST_INIT_ procedure or if the workspace structure has become corrupted.

-10 HIST_BAD_FORMAT_CALL

Nothing to format. This error can occur if you call HIST_FORMAT_ again after a previous call returned zero indicating no more text for the current context.

workspace input, output

INT .EXT:ref:(HISTWORKSPACE_TEMPLATE)

identifies the workspace area. One purpose of this area is to specify the format of the contents of the output text. The specific instance of the workspace must have been initialized by the HIST_INIT_ or HIST_PRIOR_ procedure. You can adjust the content and format of the display by setting the *workspace.FormatSelect* field; see “Considerations.”

text output

STRING .EXT:ref:*

is the buffer in which the output text is returned. The text has no termination character. The length of the text string is returned in *ret-val*.

limit input

INT:value

specifies the maximum length in bytes of the output line. This value must not be less than the minimum width specified by the HIST_MinWidth literal in the HHISTRY file; otherwise, an error occurs.

Considerations

- The *workspace.FormatSelect* field determines what information is reported. This field is initialized to a default value by the HIST_INIT_ procedure, but it can be changed by the caller before the first HIST_FORMAT_ call for any stack frame. Changing *FormatSelect* between successive calls to HIST_FORMAT_ without an intervening call to HIST_GETPRIOR_ has an undefined effect. The field is a bit mask formed by combining these literals:

HF_CodeSpace	Shows the code space in which the procedure resides; for example, user code (UC), user code RISC (UCr), accelerated system library (acc SL), system code RISC (SCr), system library RISC (SLr), or millicode (milli). For TNS code, it shows the code segment index (for example, UL.00). For named native shared run-time libraries (SRLs), it shows the SRL name (not its object file name).
HF_Context	Shows the RISC register contents whenever a full context is available (for example, when a signal is generated). If a full context is available for a frame in TNS or accelerated mode when the emulated TNS registers R0 through R7 are available, they are also shown.
HF_Context_TNS	Shows the emulated TNS registers R0 through R7 when available. This option is redundant if the HF_Context option is set.
HF_Gaps	Shows discontinuities. Three hyphens (---) denote a discontinuity in the calling sequence; for example, when a trap or system-generated nondeferrable signal occurred. An ellipsis (...) denotes missing procedure activation records in the chain of events. See “Protected Contexts” later under “Considerations.”
HF_LocLineIPF	For procedures executing in TNS/E native mode, shows the program counter (pc), previous stack pointer (PSP), and stack

pointer (sp), when relevant, in hexadecimal 64-bit values. HF_LocLineRISC is the TNS/R equivalent for HF_LocLineIPF.

The PSP is defined as FP plus the frame offset (the size of the stack frame). Compiler-listed variable offsets are relative to PSP.

If a larger set of IPF registers is being displayed because of the HF_Context or HF_Registers options, the HF_LocLineIPF information is redundant and is not shown.

HF_LocLineRISC — For procedures executing in TNS/R native mode, shows the program counter (pc), virtual frame pointer (VFP), frame pointer (FP), and stack pointer (sp), when relevant, in hexadecimal. HF_LocLineIPF is the TNS/E equivalent for HF_LocLineRISC.

The FP is used to access formal parameters and local variables of the procedure. It is typically the stack pointer (sp). The output identifies the pointer and displays its value for any procedure with a stack frame.

The VFP is defined as FP plus the frame offset (the size of the stack frame). Compiler-listed variable offsets are relative to VFP.

If a larger set of RISC registers is being displayed because of the HF_Context or HF_Registers options, the HF_LocLineRISC information is redundant and not shown.

HF_LocLineTNS — For procedures executing in TNS or accelerated mode, shows the values of P, E, L, and S (when available), in octal.

HF_Name — Shows the procedure name if available. If the name is not available, the action depends upon the rest of the *FormatSelect* field. If no other option is set (other than HF_Parent), HIST_FORMAT_ returns 0 and nothing is formatted for display. In this special case, you can change *FormatSelect* and call HIST_FORMAT_ again without causing a HIST_BAD_FORMAT_CALL error.

If any other option is set in *FormatSelect*, the code address is displayed instead of the unavailable name.

HF_Offset — Shows the offset from the beginning of the procedure of the current program location, if it is available and nonzero.

HF_Parent — Shows the name of the parent procedure for any subprocedure, in addition to the subprocedure name, in the form PROC.SUBPROC. This option is effective only if the

HF_Name option is also set. By itself, HF_Name causes only .SUBPROC to be shown.

This option supports code that is displaying a single context (as from a signal handler) without tracing the stack. In a stack trace, the parent name typically appears later in the trace.

HF_Registers — Shows all the RISC registers for which the value is known. In general, the complete set is known only when initiating (or continuing) a trace from a uContext structure (that is, the HIST_INIT_ procedure was called with *options*.<13:15> set to HO_Init_uContext). Otherwise, only those registers whose values are still known are displayed; these registers include the save registers (s0 through s8), which are saved by the called procedure if used, the stack pointer (sp) and program counter (pc), which revert to their original values when a procedure exits, and the global pointer (gp), which is not changed.

- These literals define combinations of other HF_* literals:

  HF_base = HF_Name + HF_Offset + HF_CodeSpace

  HF_trace = HF_base + HF_Gaps

  HF_withContext = HF_trace + HF_Context + HF_Context_TNS

  HF_full = HF_withContext + HF_Registers

- The default *FormatSelect* value set by the HIST_INIT_ procedure is HF_trace unless the HO_OneLine or HO_Init_address bit is set in the *options* parameter of the HIST_INIT_ procedure call; in that case, the default value is HF_base + HF_Parent.

- HF_withContext causes the full context to be displayed whenever a new context is available, such as at the invocation of a signal handler. HF_full causes all available state information to be displayed for native frames.

- Typically, the name, offset, and code space are displayed on one line, but that can spill onto multiple lines when necessary. Individual procedure names, subprocedure names, and SRL names longer than 65 characters are truncated. Truncated names have a greater-than sign (>) shown as the last character.

- Any output generated by the HF_LocLineTNS or HF_LocLineRISC option appears on a separate line, unless the HO_OneLine bit is set in the *options* parameter of the HIST_INIT_ call. Register displays occupy the last three to ten lines, depending upon the registers available.

- Protected contexts

  When a run-time event that requires immediate attention, such as a hardware trap, causes a signal to be generated, the uContext structure presented to the signal handler contains either one or two contexts. The primary context is a complete

record of the procedure context at the site of the event. If that site was in protected code, a secondary context contains part of the state at the site of transition into protected code.

A context is protected if the process was privileged when the signal was generated but the signal handler was installed from nonprivileged code.

For example, if a user program invokes a CALLABLE procedure (switching into privileged mode) and an attempt to access an invalid address occurs within the privileged code, the primary context is that of the invalid operation, and the secondary context is that of the user procedure at the site of the call to the CALLABLE procedure.

The secondary context is limited to those registers that, by convention, are saved and restored by all subsequent callers. Registers used as temporary values (including procedure parameters and return values) are not available; see the earlier discussion of the HF_Registers literal.

If the signal handler in the previous example performed a stack trace, calling the HIST_INIT_ procedure and specifying the options HO_Init_Here, HO_ShowProtected, and HO_NoSuppress, the trace would look something like this:

Returned Text	Explanation
HANDLER + 0x28 (UCr)	Indicates the current procedure (that is, the signal handler) and the offset.
PK_SIG_HANDLER_JACKET_ + 0x54 (SLr)	Indicates the procedure that called the handler and the offset of the call. The system procedure that called the handler is considered a transition frame and is suppressed by default.
---	Denotes a break in the calling sequence. In this case, PRIV_PROC_ did not call PK_SIG_HANDLER_JACKET_; the operating system intervened in response to a trap.
PRIV_PROC_ + 0x5C (SCr)	Indicates the procedure in which and the offset where the trap occurred. This context is the primary context in the uContext generated at the time of the trap. It is found automatically when tracing through the special jacket procedure. This context is suppressed unless the HO_ShowProtected option of HIST_INIT_ is in effect.

Returned Text	Explanation
...	Denotes one or more stack frames that are not shown; although a complete chain of procedure calls did take place, they are not all displayed. In this case, USER_PROC did not call PRIV_PROC_ directly; the activation record of the original CALLABLE procedure (and perhaps others in the chain of calls leading to the failure) was discarded as the signal was delivered to the user's handler.
USER_PROC + 0x1C0 (UCr)	Indicates the caller of the CALLABLE procedure most recently invoked from nonprivileged code. This is the secondary context in the uContext structure. It was generated by tracing the stack to the privileged boundary, before the switch to nonprivileged mode to enter the user's signal handler occurred.
AnotherProc + 0x2F8 (UCr)	Indicates the call site of USER_PROC.

If the signal handler instead passed its uContext and specified the HO_Init_uContext option to HIST_INIT_, the trace would start with PRIV_PROC_ and otherwise appear the same.

Example Code

This example code shows the way this procedure is typically called in the TNS/R environment:

```
STRUCT  hws (HISTWORKSPACE_TEMPLATE);
  .
  .
  .
error := HIST_INIT_ (hws, version1, options, context);
IF error <> HIST_OK THEN...;
DO BEGIN
   WHILE (len := HIST_FORMAT_ (hws, buffer, limit)) > 0D DO
      .
      ! Print text in buffer
      .
   END UNTIL (error := HIST_GETPRIOR_(hws)) <> HIST_OK;
IF error <> HIST_DONE THEN...;
```

Omit the DO loop and the HIST_GETPRIOR_ call to display only a single procedure state.

To specify a nondefault format selection, place an assignment to *workspace.FormatSelect* after the HIST_INIT_ call but outside the WHILE loop.

Example Traces: Case 1

These traces are produced by a procedure named xtracer called from a native signal handler. xtracer starts the trace by calling the HIST_INIT_ and HIST_FORMAT_ procedures. A SIGSEGV signal is generated in a procedure that is invoked from a CALLABLE procedure. The CALLABLE procedure is invoked from the unprivileged procedure HIST_TEST_ACTOR_.

For the first trace, the *options* parameter of HIST_INIT_ and the *FormatSelect* field of the *workspace* structure passed to HIST_FORMAT_ are set up as follows:

- *options* equals HO_Init_Here.
- *FormatSelect* equals HF_trace (the default value).

An example trace In the TNS/R environment :

```
xtracer + 0x60 (UCr)
handler + 0x170 (UCr)
...
HIST_TEST_ACTOR_ + 0x2F0 (UCr)
PROGRAM + 0x510 (UCr
```

An example trace In the TNS/E environment :

```
options      = HO_Init_Here
FormatSelect = HF_trace

xtracer + 0x110 (UCr)
handler + 0x220 (UCr)
...
HIST_TEST_ACTOR + 0x80 (UCr)
main + 0xAD0 (UCr)
_MAIN + 0x160 (UCr)
```

For the next trace:

- *options* equals HO_Init_Here + HO_ShowProtected.
- *FormatSelect* equals HF_trace (the default value).

The HO_ShowProtected option allows the resulting trace to show the procedure named doer that was trapped using the invalid address.

An example trace in the TNS/R environment:

```
xtracer + 0x60 (UCr)
handler + 0x170 (UCr)
---
doer + 0x5C (UCr)
...
HIST_TEST_ACTOR_ + 0x2F0 (UCr)
PROGRAM + 0x510 (UCr)
```

An example trace in the TNS/E environment:

```
options      = HO_Init_Here + HO_ShowProtected
FormatSelect = HF_trace

xtracer + 0x110 (UCr)
handler + 0x220 (UCr)
---
doer + 0x170 (UCr)
...
HIST_TEST_ACTOR + 0x80 (UCr)
main + 0xAD0 (UCr)
_MAIN + 0x160 (UCr)
```

For the next trace:

- *options* equals HO_Init_Here + HO_ShowProtected.
- *FormatSelect* equals HF_trace (the default value) + HF_Context.

The resulting trace shows the full context at the point of the trap and the partial context at the transition to privileged state.

An example trace in the TNS/R environment:

```
  xtracer + 0x60 (UCr)
  handler + 0x170 (UCr)
  ---
  doer + 0x5C (UCr)
    Mode=Native, Priv  pc=70002290
    $00:               at=00000000  v0=0000001F  v1=00008801  |
HI=0000267C
    $04:  a0=08008590  a1=00000000  a2=00000000  a3=80AD5F60  |
LO=B8987780
    $08:  t0=FFFFFFFF  t1=0000001E  t2=00000001  t3=0000001F  |
    $12:  t4=0000001F  t5=00000000  t6=00004801  t7=20040000  |
    $16:  s0=00000004  s1=00000001  s2=FFFFFFFF  s3=FFFFFFFF  |
    $20:  s4=FFFFFFFF  s5=FFFFFFFF  s6=FFFFFFFF  s7=FFFFFFFF  |
    $24:  t8=C4863C14  t9=00000000                            |  FP=sp
    $28:  gp=08008590  sp=5FFFFE88  s8=4FFFFE50  ra=70002288  |
VFP=5FFFFEB8
  ...
  HIST_TEST_ACTOR_ + 0x2F0 (UCr)
    $16:  s0=00000000  s1=00000001  s2=FFFFFFFF  s3=FFFFFFFF  |
    $20:  s4=FFFFFFFF  s5=FFFFFFFF  s6=FFFFFFFF  s7=FFFFFFFF  |  FP=sp
    $28:  gp=08008590  sp=4FFFFCE0  s8=4FFFFE50  pc=7000269C  |
VFP=4FFFFE38
  PROGRAM + 0x510 (UCr)
```

An example trace in the TNS/E environment:

```
options      = HO_Init_Here + HO_ShowProtected
FormatSelect = HF_trace + HF_Context

xtracer + 0x110 (UCr)
handler + 0x220 (UCr)
---
doer + 0x170 (UCr)
   Mode=Native, Priv
     pc:0x0000000070001910   rp:0x0000000070001C00
    psp:0x000000006DFDFE80   sp:0x000000006DFDFE00
    cfm:0x000000000000060E  bsp:0x000000006DF04100
    lc:0x0000000000000000   ec:0x0000000000000000  pred:0x0000000000001201
Static general registers (0:31)
000: 0000000000000000  00000000080003F0  400000000000058D E000000201508070
004: 0000000000000000  0000000000000000  0000000000000000 0000000000000000
008: 000000000000001C  00034A45F5F7F980  0000000000000514 0000000000000000
012: 000000006DFDFE00  00000000000BAD0D  FFFFFFFFE2048E00 FFFFFFFFE204B018
016: FFFFFFFFE2C40060  000000006DFDD250  FFFFFFFFE2048F38 0000000000000000
020: 000000006DFDFE30  FFFFFFFF91104080  FFFFFFFF91104080 FFFFFFFF91104080
024: 0000000000000000  0000000000000000  000000006DFDFBF0 000000006DFDFDC0
028: 0000000000000000  0000000000000004  00000000000028A0 00000000000028AC
     |
     |
     |
 ....
HIST_TREST_ACTOR + 0x80   (UCR)
Mode=Native, Priv
     pc:0x0000000070001CC0  rp:0x0000000070002810
    psp:0x000000006FFFFDF0  sp:0x000000006DFDFD80
    cfm:0x0000000000000409 bsp:0x00000006E000108
    lc:0x0000000000000000   ec:0x0000000000000000  pred:0x0000000000001201
Static general registers  (0:31)
000: 000000000000000   00000000080003F0  <unknown>         <unknown>
004: 0000000000000000  00000000080003F0  0000000000000000 0000000000000000
     |
     |
     |
     |
Stacked general registers (32:45)
032: 0000000000000001  4000000000000205  0000000070001C00 000000006DFDFE20
036: 0000000000000000  0000000000000000  0000000000000000 0000000000000000
040: 0000000000000001  0000000000000000  0000000000000000 000000006DFC32D0
044: 00000000080002F0  0000000000000000
  fpsr:0x0009804C8A70033F
  fpsr decode - traps:0x3F  sf0:0x000C  sf1:0x114E  sf2:0x004C  sf3:0x004C
Lower floating point registers (0:31)
000: 0000000000000000.0000000000000000   0000000000000FFF.8000000000000000
008: 000000000000FFFE.EAFCA11000000000   0000000000002FFF4.A0B0000000000000
010: 000000000000FFFE.EAD7C13369D14000   0000000000000FFE9.C9C0F20000000000
012: 000000000000FFFE.EAD7C6FC0D0551CA   0000000000001003E.0000000000000000
026: 0000000000000000.0000000000000000   0000000000000000.0000000000000000
028: 0000000000000000.0000000000000000   0000000000000000.0000000000000000
030: 0000000000000000.0000000000000000   0000000000000000.0000000000000000
main + 0xAD0 (UCr)
_MAIN + 0x160 (UCr)
```

For the next trace:

- *options* equals HO_Init_Here + HO_ShowProtected.
- *FormatSelect* equals HF_trace (the default value) + HF_LocLineRISC.

The resulting trace shows the pc, VFP, FP, and sp registers. Note that while most procedures use sp as FP, PROGRAM uses s8, so both s8 and sp are shown.

An example trace in the TNS/R environment:

```
xtracer + 0x60 (UCr)
  pc=0x70000B20  VFP=0x4FFFFAF0  FP=sp=0x4FFFF878
handler + 0x170 (UCr)
  pc=0x7000218C  VFP=0x4FFFFB20  FP=sp=0x4FFFFAF0
---
doer + 0x5C (UCr)     pc=0x70002290  VFP=0x5FFFFEB8  FP=sp=0x5FFFFE88
...
HIST_TEST_ACTOR_ + 0x2F0 (UCr)
  pc=0x7000269C  VFP=0x4FFFFE38  FP=sp=0x4FFFFCE0
PROGRAM + 0x510 (UCr)
  pc=0x700014B4  VFP=0x4FFFFE90  FP=s8=0x4FFFFE50  sp=0x4FFFFE38
```

An example trace in the TNS/E environment:

```
options      = HO_Init_Here + HO_ShowProtected
FormatSelect = HF_trace + HF_LocLineIPF

xtracer + 0x110 (UCr)
   pc:0x0000000070000CD0  psp:0x000000006FFFDFD0   sp:0x000000006FFFCD70
handler + 0x220 (UCr)
   pc:0x0000000070001520  psp:0x000000006FFFE050   sp:0x000000006FFFDFD0
---
doer + 0x170 (UCr)
   pc:0x0000000070001910  psp:0x000000006DFDFE80   sp:0x000000006DFDFE00
...
HIST_TEST_ACTOR + 0x80 (UCr)
   pc:0x0000000070001CC0  psp:0x000000006FFFFDF0   sp:0x000000006FFFFD80
main + 0xAD0 (UCr)
   pc:0x0000000070002810  psp:0x000000006FFFFEE0   sp:0x000000006FFFFDF0
_MAIN + 0x160 (UCr)
   pc:0x0000000070002CE0  psp:0x000000006FFFFF30   sp:0x000000006FFFFEE0
```

In the next example, `doer` was called in unprivileged state. For this trace:

- *`options`* equals HO_Init_uContext + HO_OneLine, and the address of the handler's context is passed in the *`context`* parameter of HIST_INIT_.
- *`FormatSelect`* equals HF_trace (the default value) + HF_LocLineRISC.

The HO_OneLine option causes the name and register states to appear on the same line. Only one frame is reported because HIST_GETPRIOR_ is not called.

An example in the TNS/R environment:

```
  doer + 0x5C (UCr)  pc=0x70002290  VFP=0x4FFFFCE0  FP=sp=0x4FFFFCB0
```

An example trace in the TNS/E environment:

```
  options      = HO_Init_uContext + HO_OneLine
FormatSelect = HF_trace + HF_LocLineIPF

doer + 0x170(UCr) pc:0x0000000070001910 psp:0x000000006DFDFE80sp:0x000000
006DFDFE00
```

Example Traces: Case 2

This trace results from a sequence of events similar to case 1, except that the CALLABLE procedure named caller attempts to divide by zero before invoking doer. The output does not include an ellipsis because the trap occurs in a CALLABLE procedure called by a nonprivileged procedure.

For this trace:

- *options* equals HO_Init_Here + HO_ShowProtected + HO_NoSuppress.
- *FormatSelect* equals HF_trace (the default value).

An example trace in the TNS/R environment:

```
xtracer + 0x60 (UCr)
handler + 0x170 (UCr)
PK_SIG_HANDLER_JACKET_ + 0x68 (SLr)
---
caller + 0x28 (UCr)
HIST_TEST_ACTOR_ + 0x2BC (UCr)
PROGRAM + 0x510 (UCr)
```

An example trace in the TNS/E environment:

```
options      = HO_Init_Here + HO_ShowProtected + HO_NoSuppress
FormatSelect = HF_trace

xtracer + 0xF0 (UCr)
handler + 0x220 (UCr)
$UD_S__SigHandlerJacket + 0x3B0 (SLr)
---
caller + 0x100 (SLr)
HIST_TEST_ACTOR_ + 0xB0 (UCr)
main + 0xAD0 (UCr)
_MAIN + 0x160 (UCr)
```

Example Traces: Case 3

The next sequence of examples is similar to case 1, except:

- The process is unprivileged when it generates a SIGSEGV signal.
- The signal occurs in millicode (a move-bytes operation).

In the first trace:

- *options* equals HO_Init_uContext + HO_NoSuppress, and the address of the handler's context is passed in the *context* parameter of HIST_INIT_.
- *FormatSelect* equals HF_trace (the default value).

An example trace in the TNS/R environment:

```
pc=%h7E0014C4  (Milli)
doer + 0xB0 (UCr)
HIST_TEST_ACTOR_ + 0x304 (UCr)
PROGRAM + 0x514 (UCr)
```

An example trace in the TNS/E environment:

```
 (millicode example)
options      = HO_Init_uContext + HO_NoSuppress
FormatSelect = HF_trace

copyData + 0x6C51 (Milli)
_SharedMilli_MOVB_FWD + 0x2B0 (Milli)
doer + 0x1A0 (UCr)
HIST_TEST_ACTOR + 0xC0 (UCr)
main + 0xB10 (UCr)
_MAIN + 0x160 (UCr)
```

In the next trace:

- *options* equals HO_Init_uContext + HO_NoSuppress, and the address of the handler's context is passed in the *context* parameter of HIST_INIT_.
- *FormatSelect* equals HF_trace (the default value) + HF_Context.

An example trace in the TNS/R environment:

```
pc=%h7E0014C4  (Milli)
  Mode=Native  pc=7E0014C4
  $00:               at=00000000  v0=0000001D  v1=00008801  |HI=00001F67
  $04:  a0=4FFFFCDC  a1=66666666  a2=00000000  a3=80AD5F60  |LO=D9E29E00
  $08:  t0=0000FE00  t1=00000004  t2=00000001  t3=0000001D  |
  $12:  t4=6666666A  t5=00000000  t6=00004801  t7=20040000  |
  $16:  s0=00000005  s1=00000001  s2=FFFFFFFF  s3=FFFFFFFF  |
  $20:  s4=FFFFFFFF  s5=FFFFFFFF  s6=FFFFFFFF  s7=FFFFFFFF  |
  $24:  t8=C6203C14  t9=00000000                            |
  $28:  gp=08008590  sp=4FFFFCB0  s8=4FFFFE50  ra=700022E4  |
doer + 0xB0 (UCr)
HIST_TEST_ACTOR_ + 0x304 (UCr)
PROGRAM + 0x514 (UCr)
```

An example trace in the TNS/E environment:

```
options      = HO_Init_uContext + HO_NoSuppress
FormatSelect = HF_trace + HF_Context

copyData + 0x6C51 (Milli)
  Mode=Native
    pc:0xFFFFFFFFE25145B1   rp:0xFFFFFFFFE250C2F0
    psp:0x000000006FFFFC10   sp:0x000000006FFFF560
    cfm:0x0000000000002E5C  bsp:0x000000006E0002C8
    lc:0x0000000000000000   ec:0x0000000000000000  pred:0x0000000000001FC1
Static general registers (0:31)
000: 0000000000000000  FFFFFFFFE2804BE0  4000000000000008 E000000201508070
004: 0000000000000000  0000000000000000  0000000000000000 0000000000000000
008: 000000000000001C  00034A45F5F7F980  0000000000000514 0000000000000000
012: 000000006FFFF560  00000000000BAD0D  00000000080003F0 0000000008000538
016: FFFFFFFFE250C040  000000006DFDF680  000000006DFDFFB8 0000000000000001
020: 000000006FFFF9D0  000000006DFDFFB8  0000000002C80014 000000006E000D48
024: C0000000000012AD  0000000000000000  000000006FFFFCC0 000000006FFFFBD0
       |
       |
       |
120: 0000000000000001  0000000000000001  0000000000000001 000000006FFFFC48
   fpsr:0x0009804C8A70033F
   fpsr decode - traps:0x3F  sf0:0x000C  sf1:0x114E  sf2:0x004C sf3:0x004C
Lower floating point registers (0:31)
000: 0000000000000000.0000000000000000   000000000000FFFF.8000000000000000
002: 000000000001003E.0000000000000001   000000000001003E.0000000000000001
004: 000000000001003E.0000000000000001   000000000001003E.0000000000000001
006: 000000000001003E.0000000000000001   000000000001003E.0000000000000001
008: 000000000000FFFE.EE3D1E6000000000   000000000002FFF4.A0B0000000000000
010: 000000000000FFFE.EE17BBE0BFC78000   000000000000FFE9.C9C0F20000000000
012: 000000000000FFFE.EE17C1BDE0729676   000000000001003E.0000000000000000
014: 000000000001003E.0000000000000000   000000000001003E.0000000000010830
016: 0000000000000000.0000000000000000   0000000000000000.0000000000000000
018: 0000000000000000.0000000000000000   0000000000000000.0000000000000000
020: 0000000000000000.0000000000000000   0000000000000000.0000000000000000
022: 0000000000000000.0000000000000000   0000000000000000.0000000000000000
024: 0000000000000000.0000000000000000   0000000000000000.0000000000000000
_SharedMilli_MOVB_FWD + 0x2B0 (Milli)
doer + 0x1A0 (UCr)
HIST_TEST_ACTOR + 0xC0 (UCr)
main + 0xB10 (UCr)
_MAIN + 0x160 (UCr)
```

In the next trace:

- *options* equals HO_Init_uContext + HO_NoSuppress, and the address of the handler's context is passed in the *context* parameter of HIST_INIT_.
- *FormatSelect* equals HF_trace (the default value) + HF_LocLineRISC (in the TNS/R environment) or HF_LocLineIPF (in the TNS/E environment).

An example trace in the TNS/R environment:

```
pc=%h7E0014C4  (Milli)  sp=0x4FFFFCB0
doer + 0xB0 (UCr)
  pc=0x700022E4  VFP=0x4FFFFCE0  FP=sp=0x4FFFFCB0
HIST_TEST_ACTOR_ + 0x304 (UCr)
  pc=0x700026B0  VFP=0x4FFFFE38  FP=sp=0x4FFFFCE0
PROGRAM + 0x514 (UCr)
  pc=0x700014B8  VFP=0x4FFFFE90  FP=s8=0x4FFFFE50  sp=0x4FFFFE38
```

An example trace in the TNS/E environment:

```
(millicode example)
options      = HO_Init_uContext + HO_NoSuppress
FormatSelect = HF_trace + LocLineIPF

copyData + 0x6C51 (Milli)
   pc:%h0000000000000000  psp:0x000000006FFFFC10   sp:0x000000006FFFF560
_SharedMilli_MOVB_FWD + 0x2B0 (Milli)
   pc:%h0000000000000000  psp:0x000000006FFFFD00   sp:0x000000006FFFFC10
doer + 0x1A0 (UCr)
   pc:0x0000000070001960  psp:0x000000006FFFFD80   sp:0x000000006FFFFD00
HIST_TEST_ACTOR + 0xC0 (UCr)
   pc:0x0000000070001D60  psp:0x000000006FFFFDF0   sp:0x000000006FFFFD80
main + 0xB10 (UCr)
   pc:0x00000000700028B0  psp:0x000000006FFFFEE0   sp:0x000000006FFFFDF0
_MAIN + 0x160 (UCr)
   pc:0x0000000070002D40  psp:0x000000006FFFFF30   sp:0x000000006FFFFEE0
```

Example Traces: Case 4

This example shows a stack trace started from a TNS procedure. In this case, the TNS procedure is a static function named xtracer in the C source file named SHTC. xtracer calls the native HIST_* procedures to perform the trace. These examples are identical in the TNS/R and TNS/E environments

- *options* equals HO_Init_Here.
- *FormatSelect* equals HF_trace (the default value).

```
SHTC.xtracer + %37 (UC.00)
main + %740 (UC.00)
_MAIN + %32 (UC.00)
```

For the next trace:

- *options* equals HO_Init_Here.
- *FormatSelect* equals HF_trace (the default value) + HF_LocLineTNS.

```
SHTC.xtracer + %37 (UC.00)
  P=%001314  E=%000200:T,UC.00  L=%023502  S=%024142
main + %740 (UC.00)
  P=%001071  E=%000200:T,UC.00  L=%023453
_MAIN + %32 (UC.00)
  P=%000125  E=%000200:T,UC.00  L=%023430
```

The next trace is from an accelerated version of the same program:

- *options* equals HO_Init_Here.
- *FormatSelect* equals HF_trace (the default value) + HF_LocLineTNS.

Note that, like the S register, the accelerated program counter (pc) value is known only at the point of the HIST_INIT_ call.

```
SHTC.xtracer + %37 (acc UC.00)
  pc=%h70420D84  P=%001314  E=%000200:T,UC.00  L=%023502  S=%024142
main + %740 (UC.00)
  P=%001071  E=%000200:T,UC.00  L=%023453
_MAIN + %32 (UC.00)
  P=%000125  E=%000200:T,UC.00  L=%023430
```

HIST_GETPRIOR_ Procedure

Summary

The HIST_GETPRIOR_ procedure establishes a previous procedure call as the process context for display by the next HIST_FORMAT_ procedure call. For an overview of how HIST_INIT_, HIST_FORMAT_, and HIST_GETPRIOR_ can be used together to perform stack tracing, see the HIST_INIT_ Procedure. This procedure displays RISC register contents for TNS/R code and IPF register contents for TNS/E code.

Syntax for C Programmers

```
#include <histry.h>

short HIST_GETPRIOR_ ( NSK_histWorkspace *workspace );
```

Syntax for TAL Programmers

```
?SOURCE $SYSTEM.SYSTEM.HHISTRY

error := HIST_GETPRIOR_ ( workspace );       ! i,o
```

Parameters

error returned value

INT

indicates the outcome of the operation:

0 HIST_OK

The procedure executed successfully.

1 HIST_DONE

The procedure has reached the base of the stack trace.

-8 HIST_ERROR

The stack tracing mechanism failed.

-9 HIST_BAD_WORKSPACE

The workspace structure has an invalid version identifier. This error can occur if HIST_GETPRIOR_ is called without first calling the HIST_INIT_ procedure or if the workspace structure has become corrupted.

workspace input, output

INT .EXT:ref:(HISTWORKSPACE_TEMPLATE)

identifies the context and format of the process state to be displayed by the next HIST_FORMAT_ procedure call. On input, it must have already been initialized by a previous call to the HIST_INIT_ procedure and might have been modified by previous calls to HIST_GETPRIOR_. On output, it identifies the previous unsuppressed context. See “Considerations.”

Considerations

- Suppression of a stack frame by HIST_INIT_ or HIST_PRIOR_ is determined by a bit mask in the field *workspace.FrameSuppress*. The HIST_INIT_ procedure initializes the field to a default set of transition frames, or to zero (0) if the HO_NoSuppress option is specified in the call to HIST_INIT_.
- The procedure activations or stack frames to be displayed must exist and remain undisturbed on the stack. You must therefore use care to ensure that these procedure activations are not disturbed while the stack trace is taking place. For example, if a stack trace is initiated from the context of the caller or by using a jump buffer, the procedure that called HIST_INIT_ or the `setjmp()` function should not exit before the calls to HIST_GETPRIOR_.
- You can call HIST_GETPRIOR_ without calling HIST_FORMAT_ to skip a frame in a stack trace. For example, a procedure can generate a stack trace starting from its caller (instead of itself) by calling HIST_INIT_ with the HO_Init_Here option, and then calling HIST_GETPRIOR_ before the first call to HIST_FORMAT_.

Example

```
error := HIST_GETPRIOR_ ( hws );
```

HIST_INIT_ Procedure

Summary

The HIST_INIT_ procedure initializes a process history display or stack trace. It validates parameter and establishes the context to display and from which to begin tracing.

HIST_INIT_, is used with the HIST_FORMAT_ and HIST_GETPRIOR_ procedures to provide the ability to display process state, including register contents and procedure activation history or stack traces. You do not need to be concerned with details of TNS or native architectures to use these procedures. This procedure displays RISC register contents for TNS/R code and IPF register contents for TNS/E code.

These procedures can be used to perform these:

- Display the state of the current procedure and its callers, typically for diagnostic purposes.
- Display the state of a process interrupted by a signal.
- Identify the procedure containing a specified code address.

  This use provides a service similar to that of ADDRTOPROCNAME, generalized to accommodate addresses in accelerated code and in native code.

For additional details and examples, see HIST_FORMAT_ Procedure.

Syntax for C Programmers

```
#include <histry.h>

short HIST_INIT_ ( NSK_histWorkspace *workspace
                  ,const uint32 version
                  ,const uint16 options
                  ,const void *context );
```

Syntax for TAL Programmers

```
?SOURCE $SYSTEM.SYSTEM.HHISTRY

error := HIST_INIT_ ( workspace            !i,o
                     ,version              !i
                     ,options              !i
                     ,context );           !i
```

Parameters

error returned value

INT

indicates the outcome of the operation:

0 HIST_OK

The procedure terminated normally.

1 HIST_DONE

The procedure has reached the base of the stack trace.

-3 HIST_BAD_VERSION

An invalid value was specified for the *version* parameter.

-4 HIST_BAD_OPTION

An invalid value was specified for the *options* parameter. An undefined bit was set.

-5 HIST_BAD_CONTEXT

A null value was specified for the *context* parameter. The *context* parameter must contain an address.

-6 HIST_NOT_IMPLEMENTED

A specified *options* bit is defined but not implemented.

-7 HIST_INIT_ERROR

An error occurred during an attempt to initialize the stack trace.

-8 HIST_ERROR

The stack tracing mechanism failed while attempting to trace back to the calling procedure.

-11 HIST_MISSING_HOOK

This error is not returned in the D40 RVU.

workspace input, output

INT .EXT:ref:(HISTWORKSPACE_TEMPLATE)

identifies the workspace area to be initialized by this procedure with information that establishes how the stack trace will proceed. This area must be allocated before you call this procedure. The address of this workspace area is passed to subsequent calls to the HIST_FORMAT_ and HIST_GETPRIOR_ procedures.

version input

INT(32):value

identifies the version of the structure and literal declarations used to compile the caller. `HistVersion1` is a literal value defined in HHISTRY to identify the initial version.

options input

INT:value

controls the initialization process and subsequent trace.

The type of context must be specified as one of these:

<13:15> =0 HO_Init_Here

starts a trace of the current stack with the context of this call to HIST_INIT_. The *context* parameter is ignored in this case.

1 HO_Init_uContext

uses the uContext structure whose address is passed in the *context* parameter. A uContext structure is a structure of type UCONTEXT_T that is passed to a signal handler installed by the SIGACTION_INIT_ or SIGACTION_SUPPLANT_ procedure. HIST_INIT_ initializes the trace at the point where the signal was generated. See “Protected contexts” under “Considerations” for more information.

2 HO_Init_JmpBuf

uses the context saved in a native jump buffer to start tracing at the point of a call to the SETJMP_ or SIGSETJMP_ procedure that filled the jump buffer. The address of the buffer is passed to HIST_INIT_ in the *context* parameter.

3 HO_Init_31Regs

is reserved for HP use.

4 HO_Init_Address

uses a 32-bit native address for the context. This address is passed to the HIST_INIT_ procedure in the *context* parameter.

A subsequent call to HIST_FORMAT_ returns context for that location. The address must point to a code location, but that location can contain TNS instructions or RISC instructions. To find out what is at a TNS address, you must first convert the TNS address into a RISC address. For details on address translation, see the *System Description Manual*.

10 HO_Init_FuncPtr

uses a 32-bit function pointer for the context. In the TNS/E environment, the function pointer is an Official Function Descriptor (OFD); a two-element array consisting of a 64-bit code address and a 64-bit GP address. The OFD is passed to the HIST_INIT_ procedure in the context parameter. A subsequent call to HIST_FORMAT_ returns the context for the code address that was passed through the OFD. In the TNS/R environment, the function pointer is a 32-bit native address that is also passed to the HIST_INIT_ procedure in the context parameter. When you specify the HO_Init_FuncPtr option in the TNS/R environment, it is equivalent to specifying the HO_Init_Address option. For more information on the HO_Init_Address option, see the description above. Note that a subsequent call to HIST_FORMAT_ always returns the correct context for the code location. This is true regardless of whether an OFD (TNS/E environment) or a 32-bit native address (TNS/R environment) is passed to the HIST_INIT_ procedure in the context parameter specified with the HO_Init_FuncPtr option. The HO_Init_FuncPtr option is not supported from the TNS environment.

In addition, the displayed context is affected when any combination of these bits are set to 1:

<11> HO_NoSuppress

enables the display of transition frames, including the shells by which TNS code calls native procedures, the system procedure that calls a signal handler, and some transitions within low-level system software. By default, transition frames are not displayed.

<10> HO_ShowProtected

enables the display of protected context when a signal is generated within protected code. If a signal is generated within protected code, such as in a procedure running privileged, the context at that site is preserved in the uContext structure passed to the signal handler, but it is not displayed unless this option is set.

<7> HO_OneLine

modifies some formatting options to optimize the display of information in a single output line. It does not, however, ensure that all the information fits in one line.

context input

EXTADDR:value

designates the procedure context for display. The type of context whose address appears in *context* must match the context type specified in the *options* parameter bits <13:15> (the HO_Init_* value) as follows:

If *options*.<13:15> specifies...	Then *context* must point to...
HO_Init_Here (0)	Anything (it is ignored)
HO_Init_uContext (1)	A uContext structure
HO_Init_JmpBuf (2)	A jump buffer
HO_Init_Address (4)	A code address

Considerations

- Suppression of a stack frame by HIST_INIT_ or HIST_PRIOR_ is determined by a bit mask in the field *workspace.FrameSuppress*. The HIST_INIT_ procedure initializes the field to a default set of transition frames, or to zero (0) if the HO_NoSuppress option of HIST_INIT_ is specified in the call to HIST_INIT_.

  HIST_INIT_ examines the *FrameSuppress* field at least once after initializing it. To prevent suppression of initial frames, you need to set the HO_NoSuppress option.

- On return from the HIST_INIT_ procedure, the designated procedure context is ready for examination and display. If that context is suppressed (and the HO_NoSuppress option is not specified), then the next previous unsuppressed context is available. If no unsuppressed context is available, HIST_INIT_ returns an error.

- Stack tracing

  The HIST_INIT_ procedure is used with the HIST_FORMAT_ and HIST_GETPRIOR_ procedures to perform stack tracing. HIST_INIT_ is called first to initialize a workspace and determine the context to be displayed. A double loop of HIST_FORMAT_ and HIST_GETPRIOR_ procedure calls then displays a history of context transitions: HIST_FORMAT_ provides the text for display, and HIST_PRIOR_ establishes the next context (for the previous procedure).

  The HO_Init_Address option of HIST_INIT_ does not start a stack trace. It generates only one context. However, the same loop of calls still works; the loop simply terminates after one iteration. When this option is specified, the only effective *workspace.FormatSelect* options are HF_Name, HF_Parent,

HF_Offset, or HF_CodeSpace. These are all selected by default. The user can assign a subset. For a description of *workspace.FormatSelect* options, see HIST_FORMAT_ Procedure.

- Protected contexts

  When a run-time event that requires immediate attention, such as a hardware trap, causes a signal to be generated, the uContext structure presented to the signal handler contains either one or two contexts. The primary context is a complete record of the procedure context at the site of the event. If that site was in protected code, a secondary context contains part of the state at the site of transition into protected code.

  For example, if a user program invokes a CALLABLE procedure (switching into privileged mode) and an attempt to access an invalid address occurs within the privileged code, the primary context is that of the invalid operation and the secondary context is that of the user procedure at the site of the call to the CALLABLE procedure.

  The secondary context is limited to those registers that, by convention, are saved and restored by all subsequent callers. Registers used as temporary values (including procedure parameters and return values) are not available.

  For a sample display of primary and secondary context, see HIST_FORMAT_ Procedure.

- TNS process support

  The HIST_INIT_ procedure can be called from a TNS process. However, HO_Init_Here and HO_Init_Address are the only supported context option for TNS callers. For native callers, the trace begins with the caller of HIST_INIT_, because the activation records for HIST_INIT_ and its shell ($HIST_INIT_) are suppressed, regardless of the setting of the HO_NoSuppress option.

  TNS stack frames can also be encountered if a trace is begun within native procedures and proceeds through TNS frames in a TNS process.

Example

```
error := HIST_INIT_ ( hws, vers, options, context );
```

INCREMENTEDIT Procedure

Summary

The INCREMENTEDIT procedure sets the increment to be added to successive line numbers for lines that will be added to an EDIT file without explicitly specified line numbers. Each time a file is opened by OPENEDIT or OPENEDIT_, the increment is reset to 1 (which would be specified in this procedure as 1000).

INCREMENTEDIT is an IOEdit procedure and can only be used with files that have been opened by OPENEDIT or OPENEDIT_.

Syntax for C Programmers

```
#include <cextdecs(INCREMENTEDIT)>

short INCREMENTEDIT ( short filenum
                     ,[ __int32_t increment ] );
```

- CEXTDECS (through the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
CALL INCREMENTEDIT ( filenum                    ! i
                    ,[ increment ] );           ! i
```

Parameters

filenum input

INT:value

specifies the file number of the open file for which the line number increment is to be set.

increment input

INT(32):value

specifies the increment to be added to successive line numbers for lines that will be added to the file without explicitly specified line numbers. The value should be specified as 1000 times the line number increment value. If this parameter is omitted, the value 1000 is used. The possible EDIT line numbers are 0, 0.001, 0.002, ... 99999.999.

Example

In these example, INCREMENTEDIT sets the line number increment value to 10.

```
INT(32) increment := 10000D;
       .
       .
CALL INCREMENTEDIT ( filenumber, increment );
```

Related Programming Manual

For programming information about the INCREMENTEDIT procedure, see the *Guardian Programmer's Guide*.

INITIALIZEEDIT Procedure

Summary

The INITIALIZEEDIT procedure allocates the EDIT file segment (EFS) to be used by IOEdit and initializes the data structures that it contains. If your program uses IOEdit, this procedure is called automatically by the first IOEdit procedure that your program calls. It is necessary to call INITIALIZEEDIT explicitly only when your program needs to specify a value for one or more of its parameters.

Syntax for C Programmers

```
#include <cextdecs(INITIALIZEEDIT)>

short INITIALIZEEDIT ( [ short *swapvol ]
                      ,[ short maxfiles ]
                      ,[ short errorabend ]
                      ,[ short nowait-option ] );
```

Syntax for TAL Programmers

```
error := INITIALIZEEDIT ( [ swapvol ]                ! i
                         ,[ maxfiles ] )             ! i
                         ,[ errorabend ]             ! i
                         ,[ nowait-option ] );       ! i
```

Parameters

error returned value

INT

returns a file-system error number indicating the outcome of the operation.

swapvol input

INT .EXT:ref:4

is a four-word array containing the name of the disk volume in which the EFS is to be allocated. If this parameter is omitted or if INITIALIZEEDIT is unable to allocate the EFS on the specified device, the name of the caller's swap volume is used. (The swap volume of the calling program is normally the same as its object file volume, unless the SWAP option of the TACL RUN command was invoked.) If the EFS cannot be allocated on the caller's swap volume, INTIALIZEEDIT then tries every disk volume in the system until it succeeds. For a description of what occurs if INITIALIZEEDIT is unable to allocate the EFS on any disk volume in the system, see the *errorabend* parameter.

maxfiles input

INT:value

specifies the maximum number of files that IOEdit will be able to have open at one time. If this parameter is omitted or if a value less than 30 is specified, 30 is used. If a value greater than 255 is specified, 255 is used.

errorabend input

INT:value

specifies the action that INITIALIZEEDIT is to take if it is unable to allocate the EFS on any disk volume. If *errorabend* is omitted or if 0 is specified, failure to allocate the EFS causes INITIALIZEEDIT to return an error 33 (unable to obtain I/O segment space); otherwise, it writes a message to the caller's home terminal and terminates abnormally. When INITIALIZEEDIT is called by another IOEdit procedure, -1 is specified for this parameter.

nowait-option input

INT:value

specifies whether to use double buffering for files opened for nowait I/O, as follows:

0	Don't use double buffering on any file.
1	Use double buffering on all files that the user opens for nowait I/O before calling OPENEDIT_ (or OPENEDIT).
2	Use double buffering on all files opened for nowait I/O, whether by the user or by OPENEDIT_ (or OPENEDIT).
3	Has the same effect as a value of 2.

If this parameter is omitted or if its value is none of the above, 0 is used.

For additional information, see Nowait Considerations.

Nowait Considerations

IOEdit always returns to the caller with the operation finished. In this sense, it does not perform nowait I/O. Note that for write operations, the operation is considered finished when the data is in the IOEdit buffer and might not yet have been passed to the file system. The *nowait-option* parameter controls how this buffering is done.

- If *nowait-option* is set to 0 or is left unspecified, and nowait I/O was specified when the file was opened, IOEdit calls AWAITIOX after every operation. You do not need to call the COMPLETEIOEDIT procedure.
- If *nowait-option* is set to 1 and IOEdit determines that the file is being accessed sequentially, IOEdit reads ahead or writes behind when it performs nowait I/O on the buffers for files that are opened for nowait access.
- If *nowait-option* is set to 2 or 3 and IOEdit determines that the file is being accessed sequentially, IOEdit reads ahead and writes behind for all files, not just those opened for nowait access.
- If *nowait-option* is set to 1, 2, or 3 and the calling process calls AWAITIO[X] with the file number set to -1 (any file), you must call COMPLETEIOEDIT to let

IOEdit know when the nowait operation on the buffer has finished, even if the process is accessing nonedit files nowait.

Example

In this example, a call to INITIALIZEEDIT specifies that the EFS be allocated on $BIGVOL, and that if the EFS cannot be allocated on $BIGVOL or any other disk volume on the system, INITIALIZEEDIT should display an error message to the home terminal and terminate abnormally.

```
INT .EXT swapvol[0:3] := [ "$BIGVOL" ];
INT errorabend := -1;
      .
      .
error := INITIALIZEEDIT ( swapvol, , errorabend );
```

Related Programming Manual

For programming information about the INITIALIZEEDIT procedure, see the *Guardian Programmer's Guide*.

INITIALIZER Procedure

Summary

The INITIALIZER procedure is used to read the startup message and, optionally, to request receipt of assign and param messages sent by the starting process (which is often a TACL process). The INITIALIZER procedure optionally initializes file control blocks (FCBs) with the information read from the startup and assign messages.

Syntax for C Programmers

- Do not call INITIALIZER from a Guardian or OSS C program. You can instead obtain the startup information from the C run-time. For information on how to obtain the startup information from a Guardian C program, see the *C/C++ Programmer's Guide*.

Syntax for TAL Programmers

```
status := INITIALIZER ( [ rucb ]                ! i
                       ,[ passthru ]            ! o
                       ,[ startupproc ]         ! i
                       ,[ paramsproc ]          ! i
                       ,[ assignproc ]          ! i
                       ,[ flags ]               ! i
                       ,[ timelimit ]           ! i
                       ,[ num^fcbs ]            ! i
                       ,[ fcb^array ] );        ! i
```

Parameters

status returned value

INT

returns one of these values:

0 This is a primary process (of a potential process pair).

-1 This is a backup process, CHECKMONITOR returned (indicating that the primary failed before establishing a takeover point), and bit 12 of *flags* is 1.

rucb input

INT:ref:*

is a table containing pointers to FCBs (see “Considerations”).

passthru output

INT:ref:*

is an array where the *startupproc*, *paramsproc*, and *assignproc* procedures can return information to or receive information from the caller of INITIALIZER.

startupproc, paramsproc, assignproc input

are application-supplied message-processing procedures that INITIALIZER calls when it receives a startup message, param message, or assign message, respectively.

These procedures must be of the form:

```
PROC name ( [ rucb ]
           ,[ passthru ]
           ,[ message ]
           ,[ msglen ]
           ,[ match ] )
             VARIABLE;
```

rucb

INT:ref:*

is the run-unit control block described in the *Guardian Programmer's Guide*.

passthru

INT:ref:*

is an array where the procedure can save information for or retrieve information from the caller of INITIALIZER.

message

INT:ref:*

is the startup message, the param message, or one of the assign messages received. The maximum length of a message is 1028 bytes (including the trailing null characters).

msglen

INT:value

is the length, in bytes, of the message.

match

INT:value

is the match count. For each assign message, the FCBs (if *rucb* is passed) are searched for a logical file name matching the logical file name contained in the assign message. If a match is found, the information from the assign message is put into the FCBs, and the match count is incremented.

If this is not an assign message or if the *rucb* parameter is not passed, the match count is always 0.

flags input

INT:value

contains several fields that determine actions to be taken by INITIALIZER, as follows:

<0:10>	Must be 0
<11>	Request assign and param messages? 0 = yes, 1 = no
<12>	Abnormally end if backup takeover occurs before first primary stack checkpoint? 0 = yes, 1 = no
<13>	If 1, CALL MONITORNET (-1)
<14>	If 1, CALL MONITORCPUS (-1)

<15> If 1 and the caller is a TNS Guardian process, CALL ARMTRAP (-1,-1);

If 1 and the caller is a native Guardian process, CALL SIGACTION_INIT_ (SIG_ABORT)

The default value is 0.

timelimit input

INT(32):value

specifies how long INITIALIZER is to wait on $RECEIVE, as follows:

>= 0D *timelimit* specifies the maximum amount of time (in units of 0.01 second) that INITIALIZER is to wait on $RECEIVE.

= -1D INITIALIZER is to wait indefinitely.

< -1D INITIALIZER calls ABEND.

If this parameter is omitted, the default value 6000D (60 seconds) is used.

num^fcbs input

INT:value

specifies the number of FCBs passed in the *fcb^array* parameter. This parameter is required for native mode processes that require FCB processing by INITIALIZER. It is optional for TNS processes.

fcb^array input

WADDR:ref:*

is an array of addresses, each of which points to an FCB to be modified by INITIALIZER. This parameter is required for native mode processes that require FCB processing by INITIALIZER. It is optional for TNS processes.

Considerations

- $RECEIVE and the INITIALIZER procedure

  The INITIALIZER procedure provides a way of receiving startup, assign, and param messages without concern for details of the $RECEIVE protocol. (For information about $RECEIVE, see the *Guardian Programmer's Guide*.) INITIALIZER opens and obtains messages from $RECEIVE; calls the user-supplied procedure, passing the messages as a parameter to the procedure; and closes $RECEIVE.

  The INITIALIZER procedure waits on $RECEIVE for the amount of time specified by the *timelimit* parameter. If a startup message is not received within that time, or if any other error is detected on $RECEIVE, INITIALIZER calls ABEND. Except in rare cases, the default *timelimit* value (60 seconds) is appropriate and should be used.

- Sequential I/O (SIO) procedures and FCBs

  If the *rucb* parameter is supplied, INITIALIZER modifies FCBs based on the information supplied by the startup and assign messages. These FCBs are in the form expected by the sequential I/O procedures and can be used with the SIO procedures without change. If the application does not use the SIO procedures to access the files, but needs to use them to get startup information, the information recovered from the assign messages can be obtained from the FCBs by using the CHECK^FILE procedure. For additional about SIO procedures, see the *Guardian Programmer's Guide*.

- Assign and param messages

  Except when invoked by the backup process of a process pair, INITIALIZER reads the startup message, then optionally requests assign and param messages (see *flags*.<11>. For each assign message, the FCBs (if *rucb* is passed) are searched for a logical file name matching the logical file name contained in the assign message. If a match is found, the information from the assign message is put into the file's FCBs, and the match count is incremented.

  For proper matching of names, the "progname" and "filename" fields of the assign message must be blank-padded.

  Note that you can perform your own processing of these messages using *startupproc*, *assignproc*, and *paramproc* irrespective of whether you use FCBs.

- Calls to ABEND

  INITIALIZER calls the ABEND procedure for any errors it detects. If INITIALIZER does call ABEND, text describing the cause of the call is passed in the process deletion (ABEND) system message. The possible causes are:

  - Timeout reading $RECEIVE.
  - Invalid value specified for the *timelimit* parameter.
  - Unable to open $RECEIVE.
  - Unable to obtain process handle.
  - Unexpected message from the creator process.
  - In the backup process of a process pair, CHECKMONITOR returned and bit 12 of *flags* was equal to zero.
  - The number of FCBs specified in ALLOCATE^CBS, ALLOCATE^CBS^D00, or *num^fcbs* is incorrect, or the format of an FCB is invalid.

  For further information about the text that is passed in the process deletion (ABEND) system message, see "INITIALIZER Errors" in the *Guardian Procedure Errors and Messages Manual*.

- FCBs and native mode

In native mode, you must use the *num^fcbs* and *fcb^array* parameters to explicitly reference any FCBs that INITIALIZER modifies, for example, using file names supplied in the Startup, ASSIGN, or PARAM messages.

Related Programming Manual

For programming information about the INITIALIZER utility procedure, see the *Guardian Programmer's Guide*.

INTERPRETINTERVAL Procedure

Summary

The INTERPRETINTERVAL procedure takes a fixed variable (quad) containing a value representing a number of microseconds and converts it into a combination of days, hours, minutes, seconds, milliseconds, and microseconds. All output parameters are optional.

This procedure is similar to CONVERTPROCESSTIME except that INTERPRETINTERVAL places no limit on the timestamp value.

Syntax for C Programmers

```
#include <cextdecs(INTERPRETINTERVAL)>

__int32_t INTERPRETINTERVAL ( long long time
                             ,[ short _near *hours ]
                             ,[ short _near *minutes ]
                             ,[ short _near *seconds ]
                             ,[ short _near *milsecs ]
                             ,[ short _near *microsecs ] );
```

- CEXTDECS (through the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
days := INTERPRETINTERVAL ( time                  ! i
                          ,[ hours ]              ! o
                          ,[ minutes ]            ! o
                          ,[ seconds ]            ! o
                          ,[ milsecs ]            ! o
                          ,[ microsecs ] );       ! o
```

Parameters

days returned value

INT(32)

returns either the number of days in the interval of time (0D or greater), or an error indication of -1D if *time* is negative.

time input

FIXED:value

specifies the 4-word fixed time interval.

hours output

INT:ref:1

returns the number of hours in the interval of time (0 or greater).

minutes output

INT:ref:1

returns the number of minutes in the interval of time (0 or greater).

seconds output

INT:ref:1

returns the number of seconds in the interval of time (0 or greater).

milsecs output

INT:ref:1

returns the number of milliseconds in the interval of time (0 or greater).

microsecs output

INT:ref:1

returns the number of microseconds in the interval of time (0 or greater).

Example

```
FIXED START, FINISH;
INT(32) DAYS;
INT HRS;
INT MIN;
INT SEC;
      .
      .
DAYS := INTERPRETINTERVAL ( FINISH - START, HRS, MIN, SEC );
IF DAYS < 0D THEN ...
```

Related Programming Manual

For programming information about the INTERPRETINTERVAL procedure, see the *Guardian Programmer's Guide*.

INTERPRETJULIANDAYNO Procedure

Summary

The INTERPRETJULIANDAYNO procedure converts a Julian day number to the year, month, and day.

The Julian calendar is the integral number of days since January 1, 4713 B.C. The formal definition of the Julian day states that it starts at 12:00 (noon), Greenwich mean time (GMT).

Syntax for C Programmers

```
#include <cextdecs(INTERPRETJULIANDAYNO)>

void INTERPRETJULIANDAYNO ( __int32_t julian-day-num
                           ,short _near *year
                           ,short _near *month
                           ,short _near *day );
```

- CEXTDECS (through the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
CALL INTERPRETJULIANDAYNO ( julian-day-num          ! i
                          ,year                     ! o
                          ,month                    ! o
                          ,day );                   ! o
```

Parameters

julian-day-num input

INT(32):value

is the Julian day number to be converted. The *julian-day-num* must not be less than 1,721,120 (year 0, month 3, day 1) nor greater than 3,182,395. Values outside this range are invalid and result in -1 being returned in the *year* parameter.

year output

INT:ref:1

returns the Gregorian year (for example, 1984, 1985, and so forth).

month output

INT:ref:1

returns the Gregorian month (1-12).

day output

INT:ref:1

returns the Gregorian day of the month (1-31).

Example

```
CALL INTERPRETJULIANDAYNO ( JULIANDAYNO, YR, MN, DAY );
```

Related Programming Manual

For programming information about the INTERPRETJULIANDAYNO procedure, see the *Guardian Programmer's Guide*.

INTERPRETTIMESTAMP Procedure

Summary

The INTERPRETTIMESTAMP procedure converts a 64-bit Julian timestamp into a Gregorian (the common civil calendar) date and time of day.

Syntax for C Programmers

```
#include <cextdecs(INTERPRETTIMESTAMP)>

__int32_t INTERPRETTIMESTAMP ( long long julian-timestamp
                              ,short _near *date-and-time );
```

- CEXTDECS (through the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
ret-date-time := INTERPRETTIMESTAMP ( julian-timestamp    ! i
                                     ,date-and-time );    ! o
```

Parameters

ret-date-time returned value

INT(32)

returns the 32-bit Julian day number. A value of -1D is returned if the supplied Julian timestamp is out of range (see “Considerations”).

julian-timestamp input

FIXED:value

is a 64-bit Julian timestamp to be converted.

date-and-time output

INT:ref:8

returns an array containing the date and time of day. A value of -1 is returned in word [0] if the supplied Julian timestamp is out of range (see “Considerations”). This array has this form:

[0]	The Gregorian year	(1984, 1985, ...)
[1]	The Gregorian month	(1-12)
[2]	The Gregorian day of month	(1-31)
[3]	The hour of the day	(0-23)
[4]	The minute of the hour	(0-59)
[5]	The second of the minute	(0-59)
[6]	The millisecond of the second	(0-999)
[7]	The microsecond of the millisecond	(0-999)

Considerations

- INTERPRETTIMESTAMP checks that the Julian timestamp corresponds to a time in the range 1 January 0001 00:00 through 31 December 10000 23:59:59.999999. If the supplied value is out of range, the procedure returns a value of -1D in *ret-date-time* and -1 in *date-and-time*[0].
- For additional information on Julian timestamps, see JULIANTIMESTAMP Procedure.

Example

```
RETURN^DATE^TIME := INTERPRETTIMESTAMP (JULIAN^TIME,
                    DATE^TIME);
```

Related Programming Manual

For programming information about the INTERPRETTIMESTAMP procedure, see the *Guardian Programmer’s Guide*.

JULIANTIMESTAMP Procedure

Summary

The JULIANTIMESTAMP procedure returns either a four-word, microsecond-resolution, Julian-date-based timestamp or the number of microseconds elapsed since the last system load.

The Julian calendar is the integral number of days since January 1, 4713 B.C. The formal definition of the Julian day states that it starts at 12:00 (noon), Greenwich mean time (GMT).

Syntax for C Programmers

```
#include <cextdecs(JULIANTIMESTAMP)>

long long JULIANTIMESTAMP ( [ short type ]
                           ,[ short _near *tuid ]
                           ,[ short _near *error ]
                           ,[ short node ] );
```

Syntax for TAL Programmers

```
retval := JULIANTIMESTAMP ( [ type  ]              ! i
                           ,[ tuid  ]              ! o
                           ,[ error ]              ! o
                           ,[ node  ] );           ! i
```

Parameters

retval returned value

FIXED

is a value representing the number of microseconds since 12:00 (noon) GMT (Julian proleptic calender) January 1, 4713 B.C., unless *type* = 3. To convert this *retval* to a more usable form, use the INTERPRETTIMESTAMP procedure.

If *type* = 3, the value returned is the number of microseconds since the last system load. To convert this *retval* to a more usable form, use the INTERPRETINTERVAL procedure.

type input

INT:value

is one of these values specifying the type of time requested:

0 Current GMT
1 System-load GMT
2 SYSGEN GMT
3 Microseconds since system load

If *type* is not supplied, then *type* 0 is used. If *type* is out of range (that is, not 0, 1, 2, or 3), then a *retval* of -1F and an *error* of -1 are returned.

tuid output

INT:ref:1

is a time-update ID. This is used when calling the SETSYSTEMCLOCK procedure with relative GMT (see SETSYSTEMCLOCK Procedure).

error output

INT:ref:1

is returned only from a remote system with one exception: a value of -1 is returned when *type* is out of range.

node input

INT:value

is the system number of the remote node from which you want the timestamp. A value of -1 indicates that this parameter is not present and that the current node should be used.

Considerations

- System message -10 (SETTIME) allows processes to determine the magnitude of and the reason for a time change. For descriptions of interprocess system messages sent to processes, see the *Guardian Procedure Errors and Messages Manual*.
- A 64-bit Julian timestamp is based on the Julian Date. It is a quantity equal to the number of microseconds since 12:00 (noon) Greenwich mean time (Julian proleptic calendar) January 1, 4713 B.C. This timestamp can represent either Greenwich mean time, local standard time, or local civil time. There is no way to examine a Julian timestamp and determine which of the three times it represents.

  Procedures that work with a 64-bit Julian timestamp are COMPUTETIMESTAMP, CONVERTTIMESTAMP, INTERPRETTIMESTAMP, JULIANTIMESTAMP, and SETSYSTEMCLOCK. Where possible, it is recommended that applications use these procedures rather than the procedures that work with 48-bit timestamps.
- A 48-bit timestamp is a quantity equal to the number of 10-millisecond units since 00:00, 31 December 1974. The 48-bit timestamp always represents local civil time (wall clock time); consequently, this value is affected by standard time/daylight saving time changes and time zone differences.

  Procedures that work with a 48-bit timestamp are: CONTIME, TIME, and TIMESTAMP.

- Process creation time is initialized by calling TIMESTAMP, which returns the local civil time in centiseconds (0.01 second = 10 milliseconds) since midnight (00:00) on 31 December 1974, in an array of three words. Only the two low-order words are saved in the process control block (PCB); this is sufficient to make the unnamed process ID unique.
- The RCLK instruction ($READCLOCK in TAL) is another source of timestamps. It returns a 64-bit timestamp representing the local civil time in microseconds since midnight (00:00) on 31 December 1974. Note that this is not a Julian timestamp and therefore it is not transferable across HP systems. Applications should avoid using the RCLK instruction except where necessary.
- Process timing uses 64-bit elapsed time counters with microsecond resolution; these are also not Julian timestamps.
- There is no way to generalize about internal timing using 64-bit Julian timestamps or 48-bit timestamps. Each section of the operating system manages time using the method most appropriate for its application.
- All time and calendar units in this discussion are defined in *The Astronomical Almanac* published annually by the U.S. Naval Observatory and the Royal Greenwich Observatory.

Note. Because processor clocks are not synchronized to the nearest microsecond, values obtained from different processors might not agree.

- The value returned by JULIANTIMESTAMP(3), a count of the number of microseconds since COLDLOAD of this processor is not affected by SETTIME. Therefore the calculation:
  JULIANTIMESTAMP(0) - JULIANTIMESTAMP(1)
  (current time) - (cold load time)
  will not always match what is returned by JULIANTIMESTAMP(3).

Example

```
MY^TIME := JULIANTIMESTAMP;      ! returns the current GMT
```

Related Programming Manual

For programming information about the JULIANTIMESTAMP procedure, see the *Guardian Programmer's Guide*.

KEYPOSITION[X] Procedures (Superseded by FILE_SETKEY_ Procedure)

Summary

The KEYPOSITION[X] procedures are used to position by primary or alternate key within a structured file. However, positioning by primary key is usually done within key-sequenced files only when using this procedure; the POSITION procedure is more commonly used for positioning by primary key within relative and entry-sequenced files.

KEYPOSITION sets the current position, access path, and positioning mode for the specified file. The current position, access path, and positioning mode define a subset of the file for subsequent access.

KEYPOSITIONX supports positioning to a primary or alternate key when the key value resides in a buffer in an extended segment. Only the *key-value* parameter is different for KEYPOSITIONX; all other parameters are the same as for KEYPOSITION.

Syntax for C Programmers

```
#include <cextdecs(KEYPOSITION)>

_cc_status KEYPOSITION ( short filenum
                        ,char *key-value
                        ,[ short key-specifier ]
                        ,[ short length-word ]
                        ,[ short positioning-mode ] );

#include <cextdecs(KEYPOSITIONX)>

_cc_status KEYPOSITIONX ( short filenum
                         ,const char *key-value
                         ,[ short key-specifier ]
                         ,[ short length-word ]
                         ,[ short positioning-mode ] );
```

- The function value returned by KEYPOSITION[X], which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL KEYPOSITION[X] ( filenum                          ! i
                     ,key-value                        ! i
                     ,[ key-specifier ]                ! i
                     ,[ length-word ]                  ! i
                     ,[ positioning-mode ] );          ! i
```

Parameters

filenum input

INT:value

is the number of an open file where the positioning is to take place.

key-value input

STRING:ref:* (Use with KEYPOSITION)
STRING .EXT:ref:* (Use with KEYPOSITIONX)

is the address of the buffer in the stack containing the key value (KEYPOSITION) or the address of the buffer containing the key value (KEYPOSITIONX).

The *key-value* may be in the user's stack or, if KEYPOSITIONX is used, in an extended data segment. The *key-value* may not be in the user's code space.

For KEYPOSITIONX, the *key-value* address must be relative; it cannot be an absolute address. If the *key-value* is in an extended segment, the extended segment must be in use at the time of the call.

key-specifier input

INT:value

designates the key field to be used as the access path for the file:

key-specifier
- 0, or if omitted, means use the file's primary key as the access path.
- predefined key specifier for an alternate-key field, means use that field as the access path.

length-word input

INT:value

contains two values:

<0:7> *compare-length* (left byte) specifies, in bytes, the length to use for key comparisons made to decide when to stop returning records under the generic or exact positioning modes.

<8:15> *key-length* (right byte) specifies how many bytes of the *key-value* are to be searched for in the file to find the initial position.

- If *length-word* is omitted, *compare-length* and *key-length* are defined to be the length of the key (*key-specifier*) defined when the file was created. That is, if *key-specifier* is omitted or 0, *compare-length* and *key-length* are the length of the primary key. If *key-specifier* is the key specifier for an alternate key, the length of the alternate-key field is used.
- If *length-word* is 0, *compare-length* and *key-length* are also 0. This results in positioning to the beginning of the file. (Although *key-value* is still a required parameter, its value is ignored when *length-word* = 0.)
- If *key-length* = 0 and *compare-length* <> 0, file-system error 21 is returned from KEYPOSITION.
- If *key-length* <> 0 and *compare-length* = 0, *compare-length* is defined to be the minimum of *key-length* or the key length defined when the file was created.
- If *key-length* <> 0 and *compare-length* <> 0, the supplied values are used.

See “KEYPOSITION and file-system Error 21” under ”Considerations.”

positioning-mode input

INT:value

<0> if 1, and if a record with exactly the *key-length* and *key-value* specified is found, the record is skipped. If the *key-specifier* indicates a non-unique alternate key, the record is skipped only if both its alternate key and its primary key match the corresponding portions of the specified *key-value* (which should be an alternate key value concatenated with a primary key value) for *key-length* bytes (which should be the sum of the alternate and primary key lengths). This option is not supported for positioning by primary key in relative or entry-sequenced files.

<1> if 1, specifies that subsequent calls to READ or READLOCK return records in descending key order (the file is read in reverse).

<2> if 1, and if *positioning-mode*.<1> = 1 (read-reverse), specifies that positioning is performed to the last record in the set of records

matching the key criteria. If *positioning-mode*.<1> = 0, this bit is ignored.

<14:15> indicates the type of key search to perform and the subset of records obtained.

0 approximate

Positioning occurs to the first record whose key field, as designated by the *key-specifier*, contains a value equal to or greater than *key-value* for *key-length* bytes (equal to or less than when read-reverse is used).

1 generic

Positioning starts at the first record whose key field, as designated by the *key-specifier*, contains a value equal to or greater than *key-value* for *key-length* bytes (equal to or less than when read-reverse is used). Records will be accessed until one is reached whose key field does not start with a value equal to *key-value* for *compare-length* bytes.

2 exact

Positioning occurs to the first record whose key field, as designated by the *key-specifier*, contains a value of exactly *compare-length* bytes and is equal to *key-value*.

If *positioning-mode* is omitted, 0 is used.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILE_GETINFO_ or FILEINFO).

= (CCE) indicates that the KEYPOSITION was successful.

> (CCG) indicates that this is not a structured disk file.

Considerations

- The calling application process is not suspended because of a call to KEYPOSITION.
- The KEYPOSITION and KEYPOSITIONX procedures expect primary-key values for relative and entry-sequenced files to be in 4-byte form. Thus, these procedures cannot be used with format 2 files (which require keys in 8-byte form). If an attempt is made to use these procedures with format 2 files, error 581 is returned. See the FILE_SETKEY_ Procedure for information on how to perform the equivalent task with format 2 files.
- Error if incomplete nowait operations pending

  A call to the KEYPOSITION procedure is rejected with an error indication if there are any incomplete nowait operations pending on the specified file.
- Positioning on duplicate or nonexistent records

No searching of indexes is done by KEYPOSITION; therefore, a nonexistent or duplicate record is not reported until a subsequent READ, READUPDATE, WRITEUPDATE, LOCKREC, READLOCK, READUPDATELOCK, or WRITEUPDATEUNLOCK is performed.

- KEYPOSITION and disk seeks

  KEYPOSITION does not cause the disk heads to be repositioned; the heads are repositioned when a subsequent I/O call (READ, READUPDATE, WRITE, and so forth) transfers data.

- Positioning exact

  If an exact KEYPOSITION is performed, and a *compare-length* is specified which is less than that specified when the file was created, *compare-length* must match the variable key length specified when the record is entered into the file. Otherwise, a subsequent call to READ, READUPDATE, WRITEUPDATE, LOCKREC, READLOCK, READUPDATELOCK, or WRITEUPDATEUNLOCK is rejected.

- Current-state indicators after a KEYPOSITION

  Current-state indicators following a successful KEYPOSITION are:

current position	is that of the record indicated by the *key-value*, *key-specifier*, *positioning-mode*, and *key-length*, or the subsequent record if *positioning-mode*.<0> is set to l.
positioning mode	is from *positioning-mode* if it is supplied; otherwise, it is approximate mode.
compare length	is determined as follows for generic searches:

```
IF length-word.<0:7> <> 0
   THEN length-word.<0:7>
ELSE
IF length-word.<8:15> > length of key-specifier
   THEN length of key-specifier
ELSE length-word.<8:15>
```

- Positioning to the middle of a duplicate alternate key

  Positioning with an alternate key is usually done by giving the alternate key value you want. This always positions the file to the first record of the set of records that contains duplicates of the specified alternate key value.

  To position to an arbitrary record within the set of duplicate records, you can specify a *key-value* consisting of the alternate key value concatenated with the primary key value of the desired record within the set, and specify the *key-length* as the sum of the alternate key length and the primary key length (except for insertion-ordered keys).

- Saving current position

When positioning by standard (not insertion-ordered) alternate key, you can save the current file position for later access by concatenating alternate-key value and primary key values in a temporary buffer. This permits you to return to that position in a key-sequenced file; for example:

```
temporary-buffer ':='
         record.altkeyfield FOR $LEN (record.altkeyfield)
       & record.primarykey FOR $LEN (record.primarykey);
```

Use this to reposition to the same record:

```
KEYPOSITION ( filenum , temporary-buffer
             , key-specifier ,
             $LEN (record.altkeyfield) +
             $LEN (record.primarykey) ,
             positioning-mode );
```

Use this to reposition to the next record:

```
KEYPOSITION ( filenum , temporary-buffer ,
              key-specifier ,
              $LEN (record.altkeyfield) +
              $LEN (record.primarykey),
              %100000 + positioning-mode );
```

In either case, if generic positioning is desired, the generic length would be placed in the upper 8 bits of the *length-word* parameter.

This method will not work when positioning with an insertion-ordered alternate key, because the value of the timestamp portion of the alternate key is not contained in the primary record. However, you can use the SAVEPOSITION and REPOSITION procedures to save and restore the current position when positioning to an insertion-ordered alternate (or any other) key is in effect.

- Positioning to the start of a file

  To position to the first record of a key-sequenced file, you can use this call to specify a zero *length-word*:

  ```
  INT ZERO := 0;
  CALL KEYPOSITION ( FILENUM , ZERO ,, ZERO );
  ```

- Considerations for Position-to-Last Option

  The standard operation of KEYPOSITION is to position to the first record that satisfies the positioning criteria specified by *key-value*, *key-length*, *compare-length*, and *positioning-mode*. When reading a file in reverse order, however, you might want to position to the last record in the set of records matching the positioning criteria. Consider these records:

Record Number	Key Value
0	AAA
1	ABA

Record Number	Key Value
2	ABB
3	ABC
4	ACA

Following an approximate KEYPOSITION to *key-value* = “AB”, *key-length* = 2, and *positioning-mode* = read-reverse, a call to READ would return record number 1 from the set of records shown above. The same call to KEYPOSITION, but with position-to-last also specified, would result in record number 3 being returned from READ.

A similar situation arises when you read a key-sequenced file with duplicate alternate keys. When an alternate-key file allows duplicate alternate keys that are ordered by primary-key value (the standard ordering method), *key-value* should be thought of as having two parts: the alternate-key value and the primary-key value. You can specify both parts or you can specify the alternate-key value only. Consider these records:

Record Number	Alternate Key	Primary Key
0	AAA	30
1	BBB	10
2	BBB	20
3	CCC	40

Following an approximate KEYPOSITION to *key-value* = “BBB”, *key-length* = 3, and *position-mode* = read-reverse, the position would be just before record 1 and after record 0. This position results because, when *key-value* is specified as “BBB”, the primary-key part is null (the lowest possible key value). A call to READ would return record 0. The same call to KEYPOSITION, but with position-to-last specified, would result in record 2 being returned from READ.

For the primary key of relative and entry-sequenced files, the *key-value* parameter to KEYPOSITION is a 4-byte string containing a doubleword record number value. When read-reverse and approximate positioning are specified, initial positioning is performed to the first record whose record number is equal to or less than the record number passed in *key-value*. Records are returned in descending record number order from successive calls to READ. The position-to-last option has no effect (is ignored) for a KEYPOSITION to an exact record number in a relative or entry-sequenced file.

Positioning to the last record in a file with KEYPOSITION is accomplished by specifying approximate mode, read-reverse, and position-to-last in the *positioning-mode* parameter, and setting *key-length* to 0. A subsequent call to READ will return the last record in the file.

- Read Reverse and SAVEPOSITION

When saving the current position in a relative or entry-sequenced file with no alternate keys, the SAVEPOSITION procedure requires an additional three words in the *positioning buffer*, for a total of seven words when read-reverse positioning is in effect. If you have programs currently using SAVEPOSITION with a four-word *positioning buffer*, please note this change.

- Read-reverse action on current and next record pointers

  Following a call to READ when reverse-positioning mode is in effect, the *next-record-pointer* contains the record number or address which precedes the current record number or address.

  Following a read of the first record in a file (where *current- record-pointer* = 0) with reverse positioning, the *next-record- pointer* will contain an invalid record number or address since no previous record exists. A subsequent call to READ would return an "end-of-file" error, whereas a call to WRITE would return an "invalid position" error (error 550) since an attempt was made to write beyond the beginning of the file.

- KEYPOSITION and file-system error 21

  If any of these conditions are true, error 21 is returned by KEYPOSITION:

  - If the primary file is a key-sequenced file and one of these is true:
    - *key-specifier* is omitted or 0 and *key-length* is greater than the key length defined for the primary file.
    - *compare-length* is greater than *key-length*.
  - If the *key-specifier* is not zero and one of these is true:
    - *key-length* is greater than the sum of length of the alternate-key field and the length of the primary key of the file.
    - *key-length* is less than or equal to the length of the alternate-key field, and *compare-length* is greater than *key-length*.
    - *key-length* is greater than the length of the alternate-key field and the primary file is not key-sequenced, and the difference of *key-length* and *compare-length* is less than 4.
    - *key-length* is greater than the length of the alternate-key field and the primary file is not key-sequenced, and the *key-length* is less than the sum of the length of the alternate-key field and the length of the primary key of the file.

- KEYPOSITIONX error

  In addition to the errors returned from KEYPOSITION, error 22 is returned from KEYPOSITIONX in either of these cases:

  - the address of the *key-value* parameter is extended, but no segment is in use at the time of the call or the segment in use is invalid.

- the address of the *key-value* parameter is extended, but it is an absolute address and the caller is not privileged.

- Queue Files

  To read a queue file in last-in, first-out order, set *positioning-mode* <0:2> := 3. There are no alternate keys for queue files; the *key-specifier* parameter must be 0 or omitted.

  When using approximate or generic positioning, the *compare-length* and *key-length* parameters should exclude the trailing 8 bytes of the record, because this field contains a system-generated timestamp. Consequently, *length-word* is typically used with queue files.

Example

```
KEY ':=' "BROWN";
COMPARE^LEN := 5;

CALL KEYPOSITION ( INFILE , KEY , , COMPARE^LEN );
```

Related Programming Manual

For programming information about the KEYPOSITION file-system procedure, see the *Enscribe Programmer's Guide* and the *Guardian Programmer's Guide*.

8 Guardian Procedure Calls (L)

This section contains detailed reference information for all user-accessible Guardian procedure calls beginning with the letter L. Table 8-1 lists all the procedures in this section.

Table 8-1. Procedures Beginning With the Letter L

LABELEDTAPESUPPORT Procedure

Summary

This procedure is callable; it provides a way for nonprivileged programs to determine if tape label processing is enabled in the system.

Syntax for C Programmers

```
#include <cextdecs(LABELEDTAPESUPPORT)>

short LABELEDTAPESUPPORT ( [ short sysnum ] );
```

Syntax for TAL Programmers

```
retvalue := LABELEDTAPESUPPORT [ ( sysnum ) ] ;      ! i
```

Parameters

retvalue returned value

INT

1	Tape label processing is enabled.
0	Tape label processing is not enabled.
< 0	The returned value is a file-system error expressed as a negative value (that is, the returned value is equal to 0 minus the error code value).

sysnum input

INT:value

specifies the system on which the inquiry is to be conducted.

Related Programming Manual

For programming information about the LABELEDTAPESUPPORT procedure, see the *Guardian Programmer's Guide*.

LASTADDR Procedure (Superseded by ADDRESS_DELIMIT_ Procedure)

Summary

Note. This procedure cannot be called by native processes. Although this procedure is supported for TNS processes, it should not be used for new development.

LASTADDR (last address) returns the ‘G’[0] relative address of the last word in the application process’s data area. (To obtain the last extended address available, use LASTADDRX.)

Syntax for C Programmers

```
#include <cextdecs(LASTADDR)>

short LASTADDR ();
```

Syntax for TAL Programmers

```
last-addr := LASTADDR;
```

Parameters

last-addr returned value

INT

returns the ‘G’[0] relative word address of the last word in the application process’s data area.

LASTADDRX Procedure (Superseded by ADDRESS_DELIMIT_ Procedure)

Summary

Note. This procedure cannot be called by native processes. Although this procedure is supported for TNS processes, it should not be used for new development.

LASTADDRX allows user programs to check stack limits or parameter addresses. LASTADDRX returns the last extended address available in the specified relative segment. A selectable extended data segment must be currently addressable (that is, a call to USESEGMENT must have been made for this segment). You can use LASTADDR with 16-bit addresses and LASTADDRX with 32-bit addresses, so both the last address and the last extended address are available to your program to check stack limits or parameter addresses.

Syntax for C Programmers

```
#include <cextdecs(LASTADDRX)>

__int32_t LASTADDRX ( [ short seg ] );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
last-addr := LASTADDRX ( [ seg ] );        ! i
```

Parameters

last-addr returned value

INT(32)

returns the last valid extended address in the segment indicated by *seg*. If either the segment is not allocated, the segment is a flat segment, or there is a parameter error, a value of -1D is returned.

seg input

INT:value

specifies the relative segment number of the segment of interest. Valid values are:

0	User data
1	If privileged, it is system data; if not, it is user data
2	Current code
3	User code
4-1023	Selectable extended data segment. This value is the segment number portion (bits <0:14>) of the segment's address.

If this parameter is omitted, 0 is used.

Note. There are additional considerations for privileged callers.

Example

```
LITERAL FEBOUNDSERR = 22;
IF ADDR > LASTADDRX ( $HIGH(ADDR).<2:14> ) THEN
  RETURN FEBOUNDSERR;
```

LASTRECEIVE Procedure (Superseded by FILE_GETRECEIVEINFO[L]_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The LASTRECEIVE procedure is used to obtain the 4-word process ID and the message tag associated with the last message read from the $RECEIVE file. This

information is contained in the file's main-memory resident access control block (ACB). An application process is not suspended because of a call to LASTRECEIVE.

Note. To ensure that you receive valid information about the last message, call LASTRECEIVE before you perform another READUPDATE on $RECEIVE. If you received an error condition on the last message, call FILEINFO or FILE_GETINFO_ to obtain the error value before you call LASTRECEIVE.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
CALL LASTRECEIVE ( [ process-id ]                    ! o
                  ,[ message-tag ] );                ! o
```

Parameters

process-id output

INT:ref:4

returns the 4-word process ID of the process that sent the last message read through the $RECEIVE file. If the process ID is of the named form and thus in the destination control table (DCT), the information returned consists of:

[0:2]	*$process-name*
[3].<0:3>	Reserved
.<4:7>	processor number where the process is executing
.<8:15>	PIN assigned by operating system to identify the process in the processor

If the process ID is of the unnamed form and thus not in the DCT, the information returned consists of:

[0:2]	*creation-time-stamp*
[3].<0:3>	Reserved
.<4:7>	processor number where the process is executing
.<8:15>	PIN assigned by operating system to identify the process in the processor

process-id (continued)

If the process ID is of the network form, the information returned consists of:

[0].<0:7>	"\"
[0].<8:15>	System number
[1:2]	Process name

[3].<0:3>	Reserved
.<4:7>	processor number in which the process is executing
.<8:15>	PIN assigned by the operating system to identify the process in the processor

message-tag output

INT:ref:1

is used when the application process performs message queuing. *message-tag* returns a value that identifies the request message just read among other requests currently queued. To associate a reply with a given request, *message-tag* is passed in a parameter to the REPLY procedure.

The value of *message-tag* is an integer between 0 and *receive-depth* minus 1, inclusive, that is not currently being used as a message tag. When a reply is made, its associated message tag value is made available for use as a message tag for a subsequent request message.

Condition Code Settings

< (CCL) indicates that $RECEIVE is not open.

= (CCE) indicates that LASTRECEIVE was successful.

> (CCG) does not return from LASTRECEIVE.

Considerations

- The process ID that is returned by LASTRECEIVE

  The process ID returned by LASTRECEIVE following receipt of a preceding open, close, CONTROL, SETMODE, SETPARAM, RESETSYNC, or CONTROLBUF system message, or a data message, identifies the process associated with the operation. The high-order three words of the process ID will be 0 following the receipt of system messages other than the ones just named.

- Synthetic process ID

  If HIGHREQUESTERS is enabled for the calling process (either because the ?HIGHREQUESTERS flag is set in the program file or because the caller used FILE_OPEN_ to open $RECEIVE) and the last message was sent by a high-PIN process, then the returned process ID is as described above except that the value of the PIN is 255. This form of the process ID is referred to as a *synthetic* process ID. It is not a full identification of the process but it is normally sufficient for distinguishing, for example, one requester from another requester. For further details, see the *Guardian Programmer's Guide*.

- Remote opener with a long process file name

  If the calling process used FILE_OPEN_ to open $RECEIVE and did not request to receive C-series format messages, and if the last message read from $RECEIVE is from a remote process that has a process name consisting of more that five

characters, then the value of *process-id* returned by LASTRECEIVE is undefined.

Example

```
CALL LASTRECEIVE ( PROG1^ID );
```

The LASTRECEIVE procedure returns the identification of the process that sent the last message.

LOCATESYSTEM Procedure (Superseded by NODENAME_TO_NODENUMBER_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The LOCATESYSTEM procedure provides the system number corresponding to a system name and returns the logical device number of the line handler controlling the path to a given system.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
ldev := LOCATESYSTEM ( sysnum                ! i,o
                     ,[ sysname ] );         ! i
```

Parameters

ldev returned value

INT

returns one of these values:

1 : 32766	The logical device number of the network line handler that controls the current path to the system designated by *sysnum*. The logical device number has at most 15 bits of magnitude and the specified system is accessible.
32767	Indicates one of these: The line handler exists and the specified system is accessible, but the line handler logical device number exceeds 15 bits of magnitude. or The specified system is the local system, so there is no line handler logical device number to return. In either case, the system number is returned in *sysnum*.
0	The specified system does not exist.
-1	All paths to the specified system are down.
-3	Bounds error occurred on *sysname* or *sysnum*.

sysnum input, output

INT:ref:1

is the number of the system to be located unless you specify *sysname*. If you specify *sysname*, then the system number that corresponds to *sysname* returns into *sysnum*.

sysname input

INT:ref:4

if present, specifies the name of the system to be located and causes the corresponding system number to be returned in *sysnum*.

Considerations

- If the caller provides *sysname*, *sysnum* is returned the corresponding number, but if the caller omits *sysname*, the caller must supply *sysnum*.
- If the *sysname* specified does not exist, *sysnum* is set to 255.
- When retrieving a line handler logical device number that exceeds 15 bits of magnitude:

  LOCATESYSTEM uses the number 32767 to represent any logical device number whose value exceeds 15 bits of magnitude. (The value 32767 is reserved and is never used as an actual logical device number.) To retrieve logical device numbers having more than 15 bits of magnitude, replace calls to LOCATESYSTEM with calls to NODENAME_TO_NODENUMBER_.

Example

```
LDEV := LOCATESYSTEM ( SYS^NUM , SYS^NAME );
```

LOCKFILE Procedure

Summary

The LOCKFILE procedure is used to exclude other users from accessing a file (and any records within that file). The “user” is defined either as the opener of the file (identified by *filenum*) if the file is not audited—or the transaction (identified by the TRANSID) if the file is audited.

If the file is currently unlocked or is locked by the current user when LOCKFILE is called, the file (and all its records) becomes locked, and the caller continues executing.

If the file is already locked by another user, behavior of the system is specified by the locking mode. There are two “locking” modes available:

- Default—Process requesting the lock is suspended (see “Considerations”).
- Alternate—Lock request is rejected with file-system error 73. When the alternate locking mode is in effect, the process requesting the lock is not suspended (see “Considerations”).

Syntax for C Programmers

```
#include <cextdecs(LOCKFILE)>

_cc_status LOCKFILE ( short filenum
                     ,[ __int32_t tag ] );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.
- The function value returned by LOCKFILE, which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL LOCKFILE ( filenum                          ! i
              ,[ tag ] );                        ! i
```

Parameters

filenum input

INT:value

is the number of an open file that identifies the file to be locked.

tag input

INT(32):value

is for nowait I/O only. *tag* is a value you define that uniquely identifies the operation associated with this LOCKFILE.

Note. The system stores the *tag* value until the I/O operation completes. The system then returns the *tag* information to the program in the *tag* parameter of the call to AWAITIO, thus indicating that the operation completed.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILE_GETINFO_ or FILEINFO).

= (CCE) indicates that the LOCKFILE was successful.

> (CCG) indicates that the file is not a disk file.

Considerations

- Nowait and LOCKFILE

  If the LOCKFILE procedure is used to initiate an operation with a file opened nowait, it must complete with a corresponding call to the AWAITIO procedure.

- Locking modes

  - Default mode

    If the file is already locked by another user when LOCKFILE is called, the process requesting the lock is suspended and queued in a “locking” queue behind other users trying to access the file. When the file becomes unlocked, the user at the head of the locking queue is granted access to the file. If the user at the head of the locking queue is requesting a lock, it is granted the lock and resumes execution. If the user at the head of the locking queue is requesting a read, the read operation continues to completion.

- Alternate mode

  If the file is already locked by another user when the call to LOCKFILE is made, the lock request is rejected, and the call to LOCKFILE completes immediately with error 73 (“file is locked”). The alternate locking mode is specified by calling the SETMODE procedure and specifying function 4.

- Locks and open files—applies to non-audited files only

  Locks are granted on an open file (that is, file number) basis. Therefore, if a process has multiple opens of the same file, a lock of one file number excludes access to the file through other file numbers.

- Attempting to read a locked file in default locking mode

  If the default locking mode is in effect when a call to READ or READUPDATE is made to a file which is locked by another user, the caller of READ or READUPDATE is suspended and queued in the “locking” queue behind other users attempting to access the file.

Note. For non-audited files, a deadlock condition—a permanent suspension of your application—occurs if READ or READUPDATE is called by the process which has a record locked by a *filenum* other than that supplied in READ or READUPDATE. (For an explanation of multiple opens by the same process, see the FILE_OPEN_ procedure.)

- Accessing a locked file

  If the file is locked by a user other than the caller at the time of the call, the call is rejected with file-system error 73 (“file is locked”) when:

  - READ or READUPDATE is called, and the alternate locking mode is in effect.
  - WRITE, WRITEUPDATE, or CONTROL is called.

- A count of the locks in effect is not maintained. Multiple locks can be unlocked with one call to UNLOCKFILE. For example:

```
    .
CALL LOCKFILE ( FILE^A,...);     ! FILE^A becomes locked.
    .
CALL LOCKFILE ( FILE^A,...);     ! is a null operation,
                                 ! because the file is
                                 ! already locked.
                                1 A condition code
                                 ! of CCE returns.
    .
CALL UNLOCKFILE ( FILE^A,...);   ! FILE^A becomes
                                   unlocked.
    .
CALL UNLOCKFILE ( FILE^A,...);   ! is a null operation,
                                 ! because the file is
                                 ! already unlocked.
                                 ! A condition code of
                                 ! CCE returns.
```

OSS Considerations

This procedure operates only on Guardian objects. If an OSS file is specified, error 2 occurs.

Related Programming Manual

For programming information about the LOCKFILE file-procedure, see the *Enscribe Programmer's Guide* and the *Guardian Programmer's Guide*.

LOCKINFO Procedure (Superseded by FILE_GETLOCKINFO_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

LOCKINFO provides information about locks (held or pending) on a local DP2 disk volume. Each call returns information about one lock, plus as many holders/waiters as the size of the caller's buffer permits; successive calls can obtain information about all the locks for a volume, file, process, or transaction.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
error := LOCKINFO ( searchtype                ! i
                  ,searchid                   ! i
                  ,ctlwds                     ! i,o
                  ,buffersize                 ! i
                  ,buffer );                  ! o
```

Parameters

error returned value

INT

is a file-system error code indicating the outcome of the call. See “Considerations” for values that may be returned.

searchtype input

INT:value

indicates the type of lock search that is desired (see the *searchid* parameter for more details on search options).

Valid values and their uses are:

0 Return lock information for volume *searchid*. A valid DP2 disk volume name must be placed in *searchid*[0:3]. Successive calls will eventually return information for all locks on that volume.

1 Return lock information for file *searchid*. A valid file name must be passed in *searchid*[0:11]. Successive calls will eventually return information for all locks on the identified file.

2 Return information on locks for volume *searchid*[0:3], requested by the process identified by the process ID in *searchid*[4:7]. Successive calls will eventually return information for all locks on the specified volume requested by the specified process. If the process ID of a named process is passed, the name must be in uppercase characters.

3 Return information on locks for volume *searchid*[0:3], requested by the TMF transaction identified by a transid in *searchid* words [4:7]. Successive calls will eventually return information for all locks on the specified volume requested by the specified transaction.

searchid input

INT .EXT:ref:12

identifies the volume, file, volume and process, or volume and transaction for which information is to be returned. Words [0:3] of *searchid* must always contain

the name of the local DP2 disk volume that is to be searched. For the four different values of *searchtype*, these formats apply:

searchtype	[0:3]	[4:7]	[8:11]
0	volume	ignored	ignored
1	volume	subvolume	file ID
2	volume	process ID	ignored
3	volume	TRANSID	ignored

ctlwds input, output

INT .EXT:ref:4

provides information needed by the file system to return successive pieces of lock information over a sequence of calls.

On input, *ctlwds* must be set to zeros before calling LOCKINFO for the first time for one combination of *searchtype*/*searchid*. On all subsequent calls, *ctlwds* must be passed to LOCKINFO as previously returned. If it is modified in any way, error 41 may be returned.

On output, *ctlwds* returns the information LOCKINFO needs for the next call. These four words must be passed exactly as they were returned on the previous call to LOCKINFO.

buffersize input

INT:value

indicates the size, in bytes, of the buffer available for returned lock information. The minimum value is 294. The size of the buffer determines how much information can be returned on each LOCKINFO call.

buffer output

STRING .EXT:ref:*

specifies the buffer in which LOCKINFO will place the lock information. The structure of the information returned in *buffer* is described under “Considerations.”

Considerations

- Structure of returned data

  The lock information returned by one call to LOCKINFO is mapped in the user-supplied buffer using the structures on the next page.

  The structure LIB describes one locked resource, and contains the byte offset from the beginning of the buffer to the first LABINFO structure. LIB also contains the number of holder/waiter entries (LABINFO occurrences) that are returned.

This TAL example describes the detailed layout of the structure:

```
STRUCT LIB ( * );
BEGIN
  STRING
    TYPE,               ! File lock = 0, record lock = 1
    LOCKLEN;            ! Byte length of locked key (0 if
                        ! not a key-sequenced file.)
  INT
    MISC,               ! Miscellaneous flags
    FILENAME[ 0:7 ],    ! Subvol/filename of locked file
    NUMLABS;            ! Number of LABINFO entries
                        ! returned
  INT(32)
    LABOFFSET;          ! Byte offset from buffer start to
                        ! first LABINFO
  STRING
    KEYVALUE[ 0:255 ]; ! Locked key value (if LOCKLEN > 0)
  INT(32)
    RECADDR = KEYVALUE;  ! Locked record ID
                         ! (if LOCKLEN = 0)
END;
```

Definitions for the MISC word of the LIB structure (the remaining bits are reserved for future use):

```
DEFINE
   GENERIC^LOCK = MISC.<0>#;    ! If set, record lock is a
                                ! generic key lock
```

The number of LABINFO entries that can be returned to the caller of LOCKINFO depends on the size of the LIB buffer (specified in the parameter *buffersize*).

The structure LABINFO describes one lock “holder” or “waiter” of/for the locked resource described by the above LIB structure. There are LIB.NUMLABS occurrences of the LABINFO structure.

The detailed layout of LABINFO is described by this example:

```
STRUCT LABINFO ( * );
BEGIN
INT
  MISC,           ! ID type, lock and grant state
                  ! (see below)
  USERID[ 0:3 ], ! Process ID or TRANSID (see below)
  RESERVED;       ! reserved for future use.
END;
```

Definitions for the MISC word of the LABINFO structure (the remaining bits are reserved for future use):

```
DEFINE
  IDTYPE     = MISC.<0> #,  ! If set: USERID is a process
                            ! ID
  GRANTSTATE = MISC.<1:3> #,! 0 = Waiting; 1 = Granted
  INTENTFLAG = MISC.<4> #;  ! Indicates the lock is an
```

```
! intent. (an intent is a lock
! internally established by
! DP2 to prevent interference
! from file lockers.)
```

- Returned error codes

  00 Information for one locked resource and all its accessors was returned without error. More locks may exist; continue calling LOCKINFO.

  01 End of lock information for *searchtype*/*searchid*.

  02 Invalid *searchtype* (not 0, 1, 2, or 3).

  11 Lock information for the file, process, or transaction in *searchid* was not found. If any information has been returned already, it is now invalid.

  12 The lock tables in DP2 were changed between calls, so any previously returned information may be invalid. To start over, set *ctlwds* to zero and call LOCKINFO again.

  21 *buffersize* is less than the minimum.

  22 The address of *ctlwds* or *buffer* is out of bounds.

  41 Checksum error on *ctlwds*. The *ctlwds* parameter has been altered between calls to LOCKINFO or was not initialized before the first call.

  45 Information for one locked resource was returned, but the supplied buffer was too small to hold all available lock accessors information (the number of holders/waiters that could be returned is always found in LIB.NUMLABS). More locks may exist, so continue calling LOCKINFO (with *ctlwds* unchanged).

  Other file-system errors may be returned; these are documented in the *Guardian Procedure Errors and Messages Manual*.

- Obtaining lock information for remote resources

  LOCKINFO accepts the designation of a remote resource in *searchid* and attempts to obtain the information.

- High-PIN considerations

  You cannot specify the process ID of a high-PIN process in the *searchid* parameter of LOCKINFO because the identifier does not fit.

  If the holder (or waiter) of a lock is a high-PIN process, the LABINFO.USERID field returned in *buffer* contains a PIN value of 255 for that process.

- Support for HP NonStop Storage Management Foundation (SMF) objects

  The LOCKINFO procedure supports single SMF logical files but does not support entire SMF virtual volumes. If the name of a SMF logical file is supplied to this procedure, the system queries the disk process of the appropriate physical volume to obtain information about current lock holders and lock waiters on the file. If the

name of a SMF virtual volume is supplied, but not a full logical file name, an error is returned.

If you call the LOCKINFO procedure and supply the name of a physical volume, lock information is returned for any file on that volume that was opened under a SMF logical file name, but the returned file name is that of the physical file supporting the logical file.

OSS Considerations

This procedure operates only on Guardian objects. OSS files cannot have Guardian locks, so there is no information to be returned. If an OSS file is specified, error 0, indicating no error, is returned; the result is the same as calling LOCKINFO on a Guardian file that has no locks.

LOCKREC Procedure

Summary

The LOCKREC procedure excludes other users from accessing a record at the current position. The “user” is defined either as the opener of the file (identified by *filenum*) if the file is not audited—or the transaction (identified by the TRANSID) if the file is audited.

Note. LOCKREC operations cannot be used with queue files.

For key-sequenced, relative, and entry-sequenced files, the current position is the record with a key value that matches exactly the current key value. For unstructured files, the current position is the relative byte address (RBA) identified by the current-record pointer.

If the record is unlocked when LOCKREC is called, the record becomes locked, and the caller continues executing.

If the file is already locked by another user, behavior of the system is specified by the locking mode. There are two “locking” modes available:

- Default—Process requesting lock is suspended (see “Considerations”).

- Alternate—Lock request is rejected with file-system error 73. When the alternate locking mode is in effect, the process requesting the lock is not suspended (see “Considerations”).

Note. A call to LOCKFILE is not equivalent to locking all records in a file; that is, locking all records still allows insertion of new records, but file locking does not. File locks and record locks are queued in the order they are issued.

Syntax for C Programmers

```
#include <cextdecs(LOCKREC)>

_cc_status LOCKREC ( short filenum
                     ,[ __int32_t tag ] );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.
- The function value returned by LOCKREC, which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL LOCKREC ( filenum                                  ! i
               ,[ tag ] );                              ! i
```

Parameters

filenum input

INT:value

is the number of an open file that identifies the file containing the record to be locked.

tag input

INT(32):value

is for nowait I/O only. *tag* is a value you define that uniquely identifies the operation associated with this LOCKREC.

Note. The system stores the *tag* value until the I/O operation completes. The system then returns the *tag* information to the program in the *tag* parameter of the call to AWAITIO[X], thus indicating that the operation completed.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILE_GETINFO_ or FILEINFO).

= (CCE) indicates that the LOCKREC was successful.

> (CCG) indicates that the file is not a disk file.

Considerations

- Nowait and LOCKREC

  If the LOCKREC procedure is used to initiate an operation with a file opened nowait, it must complete with a corresponding call to the AWAITIO procedure.

- Default locking mode

  If the record is already locked by another user when LOCKREC is called, the process requesting the lock is suspended and queued in a “locking” queue behind other users also requesting to lock or read the record.

  When the record becomes unlocked, the user at the head of the locking queue is granted access to the record. If the user at the head of the locking queue is requesting a lock, it is granted the lock and resumes execution. If the user at the head of the locking queue is requesting a read operation, the read operation continues to completion.

- Alternate locking mode

  If the record is already locked by another user when LOCKREC is called, the lock request is rejected, and the call to LOCKREC completes immediately with file-system error 73 (“record is locked”). The alternate locking mode is specified by calling the SETMODE procedure and specifying function 4.

- Attempting to read a locked record in default locking mode

  If the default locking mode is in effect when READ or READUPDATE is called for a record that is locked by another user, the caller to READ or READUPDATE is suspended and queued in the “locking” queue behind other users attempting to lock or read the record. (Another “user” means another open *filenum* if the file is not audited, or another TRANSID if the file is audited.)

Note. For non-audited files, a deadlock condition—a permanent suspension of your application—occurs if READ or READUPDATE is called by the process which has a record locked by a *filenum* other than that supplied to READ or READUPDATE. (For an explanation of multiple opens by the same process, see the FILE_OPEN_ or OPEN procedure.)

- Selecting the locking mode with SETMODE

  The locking mode is specified by the SETMODE procedure with *function* = 4.

- A count of the locks in effect is not maintained. Multiple locks can be unlocked with one call to UNLOCKFILE. For example:

```
CALL LOCKREC ( file^a,... ); ! locks the current record
                             ! in "file^a."
  ...
CALL LOCKREC ( file^a,... ); ! has no effect since the
                             ! current record is already
                             ! locked.
  ...
CALL UNLOCKREC (file^a,...); ! unlocks the current record
                             ! in "file^a."
  ...
CALL UNLOCKREC (file^a,...); ! has no effect since the
                             ! current record is not
                             ! locked.
```

- Structured files
  - Calling LOCKREC after positioning on a nonunique key

    If the call to LOCKREC immediately follows a call to KEYPOSITION where a nonunique alternate key is specified, the LOCKREC fails. A subsequent call to FILE_GETINFO_ or FILEINFO shows that an error 46 (invalid key) occurred. However, if an intermediate call to READ is performed, the call to LOCKREC is permitted because a unique record is identified.

  - Current-state indicators after LOCKREC

    After a successful LOCKREC, current-state indicators are unchanged.

- Unstructured files
  - Locking the RBA in an unstructured file

    Record positions in an unstructured file are represented by an RBA, and the RBA can be locked with LOCKREC. To lock a position in an unstructured file, first call POSITION with the desired RBA, and then call LOCKREC. This locks the RBA; any other process attempting to access the file with exactly the same RBA encounters a “record is locked condition.” You can access that RBA by positioning to RBA-2. Depending on the process’s locking mode, the call either fails with file-system error 73 (“record is locked”) or is placed in the locking queue.

  - Record pointers after LOCKREC

    After a call to LOCKREC, the current-record, next-record, and end-of-file pointers remain unchanged.

- Ways to avoid or resolve deadlocks

  One way to avoid deadlock is to use one of the alternate locking modes that can be established by *function* 4 of the SETMODE procedure. A common method of avoiding deadlock situations is to lock records in some predetermined order.

Deadlocks can be resolved if you lock records using a nowait open and call AWAITIO with a timeout specified.

OSS Considerations

This procedure operates only on Guardian objects. If an OSS file is specified, error 2 occurs.

Related Programming Manual

For programming information about the LOCKREC file-procedure, see the *Enscribe Programmer's Guide* and the *Guardian Programmer's Guide.*

LONGJMP_ Procedure

Summary

The LONGJMP_ procedure performs a nonlocal goto. It restores the state of the calling process with context saved in a jump buffer by the SETJMP_ procedure. Control returns to the location of the corresponding SETJMP_ procedure call.

Syntax for C Programmers

```
#include <setjmp.h>

jmp_buf env;

void longjmp ( jmp_buf env
               ,int value );
```

Syntax for TAL Programmers

```
?SOURCE $SYSTEM.ZGUARD.HSETJMP

LONGJMP_ ( env                ! i
          ,value );           ! i
```

Parameters

env input

INT .EXT:ref:(JMP_BUF_TEMPLATE)

indicates the address of a jump buffer containing the process context to be restored.

value input

INT(32)

specifies the value to be returned at the destination of the long jump; that is, at the location of the corresponding SETJMP_ call. If this value is set to 0D, then 1D is returned; otherwise *value* is returned.

Considerations

- LONGJMP_ is the TAL or pTAL procedure name for the C `longjmp()` function. The C `longjmp()` function complies with the POSIX.1 standard.
- Do not call LONGJMP_ with a jump buffer that contains the signal mask that was set up by a call to the SIGSETJMP_ procedure, or the system will raise a SIGABRT signal.

  LONGJMP_ can be used with a jump buffer initialized by the SIGSETJMP_ procedure only if the call to SIGSETJMP_ does not save the signal mask.
- LONGJMP_ does not return. Normally, return is made at the location of the corresponding SETJMP_ procedure.
- The jump buffer is assumed to be valid and initialized by an earlier call to SETJMP_. If an invalid address is passed or if the caller modifies the jump buffer,

the result is undefined and could cause the system to deliver a non-deferrable signal to the process.

- If LONGJMP_ detects an error, a SIGABRT or SIGILL signal is raised (except for TNS processes).
- The jump buffer must be accessible to both the LONGJMP_ procedure call and the associated SETJMP_ procedure call.
- The procedure that invoked the corresponding call to SETJMP_ must still be active. That is, the activation record of the procedure that called SETJMP_ must still be on the stack.
- A long jump across a transition boundary between the TNS and native environments, in either direction, is not permitted. Any attempt to do so will be fatal to the process.
- A nonprivileged caller cannot jump to a privileged area. Any attempt to do so will be fatal to the process. A privileged caller, however, can execute a long jump across the privilege boundary; privileges are automatically turned off before control returns to the SETJMP_ procedure.
- As a result of optimization, the values of nonvolatile local variables in the procedure that calls SETJMP_ might not be the same as they were when LONGJMP_ was called if the variables are modified between the calls to SETJMP_ and LONGJMP_. C and pTAL programs can declare variables with the volatile type qualifier; this is the only safe way of preserving local variables between calls to SETJMP_ and LONGJMP_. Alternatively, you can make the variables global.

Example

```
LONGJMP_ ( env, value );
```

Related Programming Manual

For programming information about the LONGJMP_ procedure, see the *Guardian Programmer's Guide*.

LOOKUPPROCESSNAME Procedure (Superseded by PROCESS_GETPAIRINFO_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The LOOKUPPROCESSNAME procedure is used to obtain a description of a named process pair by its name or by its index into the local destination control table (DCT). To obtain remote process pair descriptions by index, use the GETPPDENTRY procedure.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
CALL LOOKUPPROCESSNAME ( ppd );                    ! i,o
```

Parameters

ppd input, output

INT:ref:9

on input, is either:

- the internal format process name
- the entry number in the DCT ({0:n}), where the specified value is not greater than 9215,

for the entry to be returned.

On return, *ppd* is of the form:

[0:2]	Process name of entry
[3].<0:7>	processor for primary process
.<8:15>	PIN for primary process
[4].<0:7>	processor of backup process, else 0
.<8:15>	PIN of backup process, else 0
[5:8]	*process-id* of the ancestor. Note that the *process-id* is a 4-word array where *process-id*[0:2] contains the process name or creation timestamp and *process-id*[3] contains:

[3].<0:3>	Reserved
.<4:7>	processor number where the process is executing
.<8:15>	PIN assigned by the operating system to identify the process in the processor

If the process name is not in the DCT, *ppd* is unchanged.

Condition Code Settings

< (CCL)	indicates that the specified process name is not in the directory, or that the remote system could not be accessed, or that the specified process pair has a high-PIN process as the primary or backup.
= (CCE)	indicates that the specified name was found.
> (CCG)	indicates that the specified entry number exceeds the last table entry.

Considerations

- Network use

  Remote DCT entries can be obtained by passing the process name (in network form) of the process desired. On return, the process name remains in network form.

  This is an example of using LOOKUPROCESSNAME to get the DCT entry for the name process “$PROC” running on the system “\DETROIT”:

```
EXTERNAL^NAME ':=' 17 * [ " " ];  ! blanks.
EXTERNAL^NAME ':=' "\DETROIT.$PROC";
   ! note that "$proc1" is not a valid remote name.
CALL FNAMEEXPAND ( EXTERNAL^NAME , INTERNAL^NAME , DEFAULTS
);
   ! converts \DETROIT to its system number.
CALL LOOKUPPROCESSNAME ( INTERNAL^NAME );
   ! returns the desired DCT entry.
```

  To obtain DCT entries using an *entry-num*, use the GETPPDENTRY procedure.

  If you call LOOKUPPROCESSNAME for a named process pair whose ancestor is a named process on a remote node with a process name of six characters (including the $), *ppd*[5:8] is returned filled with zeros.

- High-PIN considerations

  If you call LOOKUPPROCESSNAME for a named process pair that has a high-PIN process as the primary or backup, condition code < (CCL) is returned.

  If you call LOOKUPPROCESSNAME for a named process pair that has a high-PIN process as the ancestor, a synthetic process ID is returned in *ppd*[5:8]. A synthetic process ID contains a PIN value of 255 in place of a high-PIN value, which cannot be represented by 8 bits.

- DCT Index as an input parameter

  Although supported for backward compatibility, use of a DCT index as an input parameter is not recommended. The maximum DCT index value that LOOKUPPROCESSNAME can handle as input is 9215, which is far below the system limit. This also means that LOOKUPPROCESSNAME cannot reliably be used to scan the entire DCT by index.

- This procedure does not return information on a named process that is reserved for future use and is not started.

9 Guardian Procedure Calls (M)

This section contains detailed reference information for all user-accessible Guardian procedure calls beginning with the letter M. Table 9-1 lists all the procedures in this section.

Table 9-1. Procedures Beginning With the Letter M

MBCS_ANY_KATAKANA_ Procedure

Summary

The MBCS_ANY_KATAKANA_ procedure checks a string of HP Kanji characters for any Katakana characters. Katakana 1-byte characters are permitted in a string of HP Kanji characters.

Syntax for C Programmers

```
#include <cextdecs(MBCS_ANY_KATAKANA_)>

short MBCS_ANY_KATAKANA_ ( char *buffer
                          ,short length
                          ,[ short charset ] );
```

Syntax for TAL Programmers

```
result := MBCS_ANY_KATAKANA_ ( buffer                ! i
                              ,length                ! i
                              ,[ charset ] );        ! i
```

Parameters

result returned value

INT

returns the result of the MBCS_ANY_KATAKANA_ test. The returned value is either 0 or 1:

0 indicates that the *buffer* string does not contain any Katakana characters or that *charset* did not specify the HP Kanji multibyte character set.

1 indicates that the HP Kanji *buffer* string contains at least one Katakana character.

buffer input

STRING .EXT:ref:*

is the string to be tested for Katakana characters. The *buffer* pointer is not moved or changed by the MBCS_ANY_KATAKANA_ procedure.

length input

INT:value

is the number of bytes in the *buffer* string. The MBCS_ANY_KATAKANA_ procedure tests only the number of bytes specified in the *length* parameter and does not access the area beyond *buffer*[*length* - 1].

charset input

INT:value

identifies the multibyte character set (MBCS) to be used. If *charset* is omitted or null, the default MBCS character set identifier returned from the MBCS_DEFAULTCHARSET_ procedure is used. The presence of Katakana characters is not valid in conjunction with any MBCS other than HP Kanji.

Any value may be specified; however, the returned *result* will always be 0 if 1 (for HP Kanji) is not specified.

Considerations

The Japanese 1-byte Kanji character set is defined in the Japanese Industrial Standard (JIS) X0208 (formerly C6226); the Japanese Katakana character set is defined in JIS X0201 (formerly C6220).

MBCS_CHAR_ Procedure

Summary

The MBCS_CHAR_ procedure indicates whether a string of bytes is part of an HP multibyte character set (MBCS) and that *testmbcschar* points to the first byte of a valid character of *charset* MBCS.

The MBCS_CHAR_ procedure also performs a positive range test on all bytes of the referenced character. If all bytes pass the range test, TRUE is returned, otherwise FALSE is returned.

Syntax for C Programmers

```
#include <cextdecs(MBCS_CHAR_)>

short MBCS_CHAR_ ( char *testmbcschar
                  ,[ short charset ]
                  ,[ short *charinfo ] );
```

Syntax for TAL Programmers

```
result := MBCS_CHAR_ ( testmbcschar          ! i
                      ,[ charset ]           ! i
                      ,[ charinfo ] );       ! i,o
```

Parameters

result returned value

INT

returns the result of the MBCS character test.

0	indicates that *charset* is not a supported MBCS (see *charinfo* parameter), or *charset* is a supported MBCS and *testmbcschar* does not point to the first byte of a valid character of one of the MBCS character sets listed under *charset*.
nonzero	indicates that the character set is a supported MBCS, and *testmbcschar* points to the first byte of a valid character of *charset* MBCS. For 2-byte character sets, the returned value is the integer value of the sixteen bits which form the multibyte character, using byte-1 as the high order byte and byte-2 as the low order byte of the pair. All currently supported MBCSs are 2-byte character sets.

testmbcschar input

STRING .EXT:ref:*

is an extended pointer to the first of a group of bytes to be tested for membership in the MBCS identified by the *charset* parameter. The caller is responsible for ensuring legitimate access to all bytes of the group. All bytes are range tested for valid membership in the specified character set. If any byte fails the range test, the group fails and FALSE is returned. The *testmbcschar* pointer is not altered by the MBCS_CHAR_ procedure.

charset input

INT:value

identifies the MBCS to be used. If *charset* is omitted or null, the default MBCS identifier returned from the MBCS_DEFAULTCHARSET_ is used.

These MBCSs are supported by the MBCS_CHAR_ procedure:

1 HP Kanji
9 HP Hangul
10 HP Chinese Big 5
11 HP Chinese PC
12 HP KSC5601

charinfo input, output

INT .EXT:1

on input, *charinfo* specifies the number of bytes, beginning with *testmbcschar*, that may be read by the MBCS_CHAR_ procedure. If *charinfo* is equal to or greater than the size of a single multibyte character of the MBCS identified by *charset*, the MBCS_CHAR_ procedure tests for the presence of a multibyte character. If the integer value supplied in *charinfo* is less than the size of a single multibyte character of the MBCS identified by *charset*, no test is made and FALSE with no error indication is returned. When omitted, it is assumed that at least enough bytes can be read to compose a single multibyte character of the MBCS identified by *charset*. There is no null value for this parameter.

If *charinfo* is returned on output, it provides this information:

when *result* is nonzero:

<0:7> contains the display size (in columns) of the multibyte character identified by the test.

<8:15> contains the internal size (in bytes) of the multibyte character identified by the test.

when *result* is zero, *charinfo* contains one of these values indicating the cause of failure of the MBCS_CHAR_ test:

0 No reported error; tested character is a 1-byte character
29 Required parameter missing
-2 An unknown character set was specified

Considerations

- Tests are provided for HP Kanji (Shift-JIS), HP Hangul, HP Chinese Big 5, HP Chinese PC and HP Korean KSC5601 format MBCS. HP Kanji is the standard internal representation used by HP for the character set defined in the JIS X0208 standard (formerly JIS C6226). Chinese Big 5 is a character set defined by vendors in Taiwan. Chinese PC is the character set used by IBM on Chinese PCs. HP Hangul support is provided for the Korean character set in use by KIPS on HP

6526 terminals as well as for the new standard Hangul (KSC 5601) character set. All MBCSs are similar in format and are suitable for internal representation of the multibyte character set in conjunction with an ASCII-like 1-byte character set.

- In most supported MBCS schemes, single-byte values have multiple- character identities. For example, in the HP Kanji MBCS format, all ASCII alphabetic and all 1-byte Japanese Katakana characters also appear within HP Kanji MBCS characters. Furthermore, all byte values which appear as the first byte of HP Kanji characters might also appear in the second byte of HP Kanji characters. Similar ambiguous usage of individual byte values occurs in other supported MBCSs. Proper character identification depends both on value range testing and context. Because of the multiple identity of individual byte values, it is not safe to attempt to identify characters selected at random from a text string. Proper character identification requires analysis of text strings from a starting location with known conditions. Character analysis must begin on a byte position that is known to be either a 1-byte character or the first byte of a multibyte character.

- To obtain correct results, supply a valid starting point and ensure legitimate access to the text buffer. Text strings should begin only with a 1-byte character or with the first byte of a multibyte character. Thus, you can call the MBCS_CHAR_ procedure with the *testmbcschar* parameter set to the address of the first byte of a text string. Subsequent calls to test other locations within the text string must be based upon the results of the initial and succeeding calls, with the *testmbcschar* pointer being advanced by the size of the character found, following each call to MBCS_CHAR_. This code sample illustrates the proper use of the MBCS_CHAR_ procedure:

```
@testmbcschar := @first byte in text string;
WHILE processing mixed text string
DO
BEGIN    --text string loop
charsize := number of bytes remaining in text string;
IF MBCS_CHAR_( testmbcschar, charset, charsize )
THEN -- found valid MBCS character
    BEGIN    -- process and advance pointer

    ... user-required MBCS character processing here ...

    @testmbcschar := @testmbcschar +
                     $dbl(charsize.<8:15>);
    END    -- process and advance pointer
ELSE -- found a 1-byte character
    BEGIN    -- process and advance pointer

    ... user-required 1-byte character processing here ...

    @testmbcschar := @testmbcschar + 1d;
    END;    -- process and advance pointer
END;        -- text string loop
```

When calling the MBCS_CHAR_ procedure, you must prevent attempts to read out-of-bounds data. In the preceding example, the amount of remaining buffer

space (number of bytes) is conveyed by the *charinfo* parameter on the call; MBCS_CHAR_ does not attempt to access data beyond this buffer. When this parameter is omitted, the MBCS_CHAR_ procedure operates upon the assumption that enough bytes may be read to compose one character of the current MBCS. The caller assumes responsibility for the accuracy of this assumption.

Related Programming Manual

For programming information about the MBCS_CHAR_ procedure, see the *Guardian Programmer's Guide*.

MBCS_CHARSIZE_ Procedure

Summary

The MBCS_CHARSIZE_ procedure returns the display size (in columns) and the storage size (in bytes) of multibyte character set (MBCS) characters from the character set specified by the *charset* parameter.

The storage size of all supported internal MBCSs has a 1:1 relationship to the number of display columns required; thus, a 20-byte string of HP Kanji characters requires 20 columns of display space on a terminal or printer.

Syntax for C Programmers

```
#include <cextdecs(MBCS_CHARSIZE_)>

short MBCS_CHARSIZE_ ( [ short charset ] );
```

Syntax for TAL Programmers

```
result := MBCS_CHARSIZE_ [ ( charset ) ];       ! i
```

Parameters

result returned value

INT

is the size in bytes of each character in the MBCS specified by the *charset* parameter.

0	indicates that either no MBCS is configured or the specified MBCS is not supported.
nonzero	indicates that the *result* parameter contains this information:
<0:7>	contains the display size (in columns) of the multibyte character identified by the test.
<8:15>	contains the internal size (in bytes) of the multibyte character identified by the test.

charset input

INT:value

identifies the multibyte character set (MBCS) to be used. If *charset* is omitted or null, the default MBCS identifier returned from the MBCS_DEFAULTCHARSET_ is used. These MBCSs are supported by the MBCS_CHAR_ procedure:

1 HP Kanji
9 HP Hangul
10 HP Chinese Big 5
11 HP Chinese PC
12 HP KSC5601

Related Programming Manual

For programming information about the MBCS_CHARSIZE_ procedure, see the *Guardian Programmer's Guide*.

MBCS_CHARSTRING_ Procedure

Summary

The MBCS_CHARSTRING_ procedure tests the contents of a data string for the exclusive use of MBCS characters of known internal character sets. This procedure depends upon the MBCS_CHAR_ procedure to test each group of bytes in the data string for validity; it inherently supports all the character sets known to the MBCS_CHAR_ procedure. The MBCS_CHARSTRING_ procedure recognizes blank MBCS characters. For the purposes of this procedure, a blank MBCS character is a string of blank (%H20) bytes of the same storage length as an MBCS character of the current MBCS.

Syntax for C Programmers

```
#include <cextdecs(MBCS_CHARSTRING_)>

short MBCS_CHARSTRING_ ( char *testmbcschar
                        ,short bytecount
                        ,short *index
                        ,[ short charset ]
                        ,[ short *charinfo ] );
```

Syntax for TAL Programmers

```
result := MBCS_CHARSTRING_ ( testmbcsstring          ! i
                            ,bytecount               ! i
                            ,index                   ! o
                            ,[ charset ]             ! i
                            ,[ charinfo ] );         ! o
```

Parameters

result returned value

INT

returns the *result* of the MBCS character test of the *testmbcsstring* text string.

0 indicates that the *charset* parameter contains an unknown MBCS identifier (see *charinfo* description) or contains a known MBCS identifier but the test of *testmbcsstring* for valid characters failed.

1 indicates that all MBCS characters in the *testmbcsstring* are valid characters (or blanks) of the specified MBCS.

testmbcsstring input

STRING .EXT:ref:*

is an extended pointer to the first byte of a data string to be tested. The contents of the data string are not altered by the MBCS_CHARSTRING_ procedure.

bytecount input

INT:value

is the number of bytes contained in *testmbcsstring*. The MBCS_CHARSTRING_ procedure tests only the number of bytes specified in the *bytecount* parameter and does not access the area beyond *testmbcsstring*[*bytecount*-1].

index output

INT .EXT:ref:1

is the byte index of the first byte group found in the string that is not a valid MBCS character and is not a group of blanks (%H20) the size of an MBCS character.

charset input

INT:value

identifies the MBCS to be used. If *charset* is omitted or null, the default character set from the MBCS_DEFAULTCHARSET_ procedure is used. The MBCS_CHARSTRING_ procedure does not examine or validate the character set identification, but simply passes it on to the MBCS_CHAR_ procedure. MBCS_CHARSTRING_ inherently supports all the MBCSs known to the MBCS_CHAR_ procedure.

charinfo output

INT .EXT:ref:1

indicates the cause of failure of the MBCS_CHARSTRING_ test. The MBCS_CHARSTRING_ procedure returns a file-system error 29 to indicate missing required parameters; other error indications are passed back from the MBCS_CHAR_ procedure. For returned values and interpretations, see MBCS_CHAR_ Procedure.

Considerations

The MBCS_CHARSTRING_ procedure uses the MBCS_CHAR_ procedure to test the specified text string for multibyte characters. All MBCSs supported by the MBCS_CHAR_ procedure are inherently supported by the MBCS_CHARSTRING_ procedure.

MBCS_CODESETS_SUPPORTED_ Procedure

Summary

The MBCS_CODESETS_SUPPORTED_ procedure returns a 32-bit integer value. Each bit of the returned value indicates the presence of a particular multibyte character set (MBCS).

Syntax for C Programmers

```
#include <cextdecs(MBCS_CODESETS_SUPPORTED_)>

__int32_t MBCS_CODESETS_SUPPORTED_ ();
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
result := MBCS_CODESETS_SUPPORTED_ ;
```

Parameters

result returned value

INT(32)

is a 32-bit value indicating available MBCS support. A bit set to 1 indicates the presence of an MBCS:

<1> = HP Kanji
<2> = IBM Kanji
<3> = IBM Kanji Mixed
<4> = JEF (Fujitsu) Kanji
<5> = JEF (Fujitsu) Kanji Mixed
<6> = reserved
<7> = JIS Kanji
<8> = reserved
<9> = HP Hangul
<10> = Chinese Big 5
<11> = Chinese PC (5550C)
<12> = HP KSC5601 (with KIPS extensions)

Other bits are unassigned.

Related Programming Manual

For programming information about the MBCS_CODESETS_SUPPORTED_ procedure, see the *Guardian Programmer's Guide*.

MBCS_DEFAULTCHARSET_ Procedure

Summary

The MBCS_DEFAULTCHARSET_ procedure returns the default multibyte character set (MBCS) identification.

HP systems support various MBCSs in different ways. HP Kanji (Shift-JIS), Chinese Big 5, Chinese PC, Hangul, and KSC5601 data formats are supported as internal code representations. IBM and Fujitsu Kanji formats are supported by translation from the HP Kanji internal format.

The MBCS_DEFAULTCHARSET_ procedure returns the default MBCS internal format in use on the system queried. The default value is hardcoded; that is, it can be changed only by reconfiguring the system.

Note. Each system must have a MBCS_DEFAULTCHARSET_ specified.

Syntax for C Programmers

```
#include <cextdecs(MBCS_DEFAULTCHARSET_)>

__int32_t MBCS_DEFAULTCHARSET_ ();
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
result := MBCS_DEFAULTCHARSET_ ;
```

Parameters

result returned value

INT

returns the identifier of the default MBCS character set:

```
 0 = No MBCS configured
 1 = HP Kanji
 9 = HP Hangul
10 = HP Chinese Big 5
11 = HP Chinese PC
12 = HP KSC5601
```

Considerations

HP Kanji is the default character set. This default can only be changed by reconfiguring the system. Contact your HP representative for information on changing the default MBCS.

Related Programming Manual

For programming information about the MBCS_DEFAULTCHARSET_ procedure, see the *Guardian Programmer's Guide*.

MBCS_EXTERNAL_TO_TANDEM_ Procedure

Summary

The MBCS_EXTERNAL_TO_TANDEM_procedure translates a text string from a specified external format to the HP internal text format.

Syntax for C Programmers

Note. In the TNS/E environment the CEXTDECS file uses the *int* data type for 32-bit values. This is a change from the TNS and TNS/R environments where CEXTDECS uses the *long* data type for 32-bit values.

```
#include <cextdecs(MBCS_EXTERNAL_TO_TANDEM_)>

short MBCS_EXTERNAL_TO_TANDEM_ ( __int32_t *source-string
                               ,__int32_t *destination- string
                                 ,short source-length
                                 ,short maximum-length
                                 ,short intermediate
                                 ,short external-form
                                 ,short *finished-length
                                 ,[ char *shift-to-MBCS ]
                               ,[ char *shift-to-one-byte ] );
```

Syntax for TAL Programmers

```
error-code := MBCS_EXTERNAL_TO_TANDEM_
                            ( source-string              ! i,o
                             ,destination-string         ! i,o
                             ,source-length              ! i
                             ,maximum-length             ! i,o
                             ,intermediate               ! i
                             ,external-form              ! i
                             ,finished-length            ! o
                             ,[ shift-to-MBCS ]          ! i
                             ,[ shift-to-one-byte ] ); ! i
```

Parameters

error-code returned value

INT

returns a procedure error code. Possible error codes are:

0 Successful completion of the translation
-1 Translation truncated due to lack of destination buffer space
-2 Unknown translation requested
-3 Invalid source string length
-4 Invalid character in Kanji-only source string
-5 Control string parameter too long
29 Required parameter missing

source-string input, output

INT(32) .EXT:ref:1

is a pointer to a double-word integer containing the extended address of the source text string to be translated. After translation, the address points to the byte following the last byte in the source string that was successfully translated.

destination-string input, output

INT(32) .EXT:ref:1

is a pointer to a double-word integer containing the extended address of the location to receive the translated text string. After translation, the address points to the byte following the last byte in the destination string.

source-length input

INT:value

is the length, in bytes, of the source text string.

maximum-length input, output

INT .EXT:ref:1

on input, is the maximum allowable number of bytes of space in the output destination string.

While all formal parameters (except *shift-to-MBCS* and *shift-to-one-byte*) are mandatory for string translation, specifying only the *source-length*, *maximum-length*, and *external-form* parameters (omitting all other parameters), returns the maximum length required for the destination string, without any string translation.

intermediate input

INT:value

is a logical flag indicating the optional forms of the source data string.

For translations from an EBCDIC type of data format, this parameter is interpreted as follows:

When TRUE, the source text string is in an intermediate form which must be further processed to yield the correct ASCII/JIS format for one-byte characters. An example of this is data from IBM Katakana devices which has already been through the HP “universal” EBCDIC/ASCII conversion.

When FALSE, the source data string is still in EBCDIC format. It has not been through the HP universal EBCDIC/ASCII conversion.

For translations from a format containing JIS standard Kanji and a JIS or ASCII-like one-byte character set, this parameter is interpreted as follows:

When TRUE, the source text stream is in shift-in/ shift-out (SI/SO) format. Conversion of the source text stream begins in shift-in state. ASCII SI/SO characters that frame data character sub-strings are removed and each byte

whose byte value is greater than octal 40, in the SI/SO framed substrings, has the high-order bit turned on.

When FALSE, the source text stream is in eight-bit data format. SI/SO processing is not done.

external-form input

INT:value

indicates the format of the source text stream:

- IBM external formats
  - Data stream without substring frames

    0 IBM Kanji only (without subfield strings)
  - Data stream using SO/SI substring frames

    1 IBM Kanji EBCDIC

    2 IBM Kanji/Katakana-EBCDIC
  - Data stream using character attribute substring framing (IBM 3270 data stream only)

    11 IBM Kanji EBCDIC

    12 IBM Kanji/Katakana-EBCDIC
- JEF external formats
  - Data stream using KI/KO substring frames

    3 JEF (Fujitsu) Kanji only

    4 JEF (Fujitsu) Kanji EBCDIC

    5 JEF (Fujitsu) Kanji/Katakana-EBCDIC
- Other external formats

    8 JIS X0208 Kanji/JIS X0201 (was C6226/C6220)

finished-length output

INT .EXT:ref:1

contains the byte count of the translated destination string upon successful completion of the translation. For other cases, the value is undefined.

shift-to-MBCS input

STRING .EXT:ref:*

is an optional pointer to a string containing the escape or control string used to indicate a shift to a multibyte character set in the source text string. This string

must always be in HP internal (not EBCDIC) character format, regardless of the final form of the source string.

This procedure does not contain logic for identifying escape or control strings. The control strings that are used are either the specified default control strings or user-supplied alternative control strings. In earlier versions of this procedure, control strings were null delimited. While null-delimited control strings are still supported, an alternative format that supports control strings containing null bytes is now also provided. The IBM Kanji character attribute (described following) is an example of an alternative format.

The alternative format must be expressed as (null flag, count, string), where the first byte is a null flag indicator of the alternative string form, the second byte contains an integer value representing the length of the control string, and the third and subsequent bytes represent the value of the control string.

The minimum length for a control string is 1 byte, and the maximum length is 20 bytes. If the procedure receives a zero-length control string, the results are undefined.

When the control string values are not user-supplied, the default values used for control strings expressed in the original null-delimited format are as follows:

IBM Kanji	[%H0E,null]
JEF (Fujitsu) Kanji	[%H88,null]
JIS X0208 Kanji	[%H1B, %H24,%H42,null]

When the control string values are not user-supplied, the default values used for control strings expressed in the alternative format are as follows:

IBM Kanji character attribute	[null, %H03, %H88, %HA2, %H38]

shift-to-one-byte input

STRING .EXT:ref:*

is an optional pointer to a string containing the escape or control string used to indicate a shift to a 1-byte character set in the source text string. This string must always be in HP internal (not EBCDIC) character format, regardless of the final form of the source string.

This procedure does not contain logic for identifying escape or control strings. The control strings that are used are either the specified default control strings or user-supplied alternative control strings. In earlier versions of this procedure, control strings were null delimited. While null-delimited control strings are still supported, an alternative format that supports control strings containing null bytes is now also provided. The IBM Kanji character attribute (described following) is an example of an alternative format.

The alternative format that supports user-supplied control strings containing null bytes must be expressed as (null flag, count, string), where the first byte is a null flag indicator of the alternative string form, the second byte contains an integer value representing the length of the control string, and the third and subsequent bytes represent the value of the control string.

The minimum length for a control string is 1 byte, and the maximum length is 20 bytes. If the procedure receives a zero-length control string, the results are undefined.

When the control string values are not user-supplied, the default values used for control strings expressed in the original null-delimited format are as follows:

IBM Kanji	[%H0F,null]
JEF (Fujitsu) Kanji	[%H89,null]
JIS X0208 Kanji	[%H18,%H28,%H4A,null]

When the control string values are not user-supplied, the default values used for control strings expressed in the alternative format are as follows:

IBM Kanji character attribute	[null, %H03, %H88, %HA2, %H00]

Considerations

- All parameters except *shift-to-MBCS* and *shift-to-one-byte* are necessary for a string translation operation.
- To determine the maximum length of the destination string, you must specify the *source-length*, *maximum-length*, and *external-form* parameters. The other formal parameters can be omitted if you want to determine only the maximum length of the destination string, without performing any string translation. When performing string translation, if less than this amount of space is allowed, the translation procedure might fail due to insufficient space in the destination string.
- Any invalid 2-byte character that is found in the source string is mapped to the value %HFCFC. Any nondisplayable 2-byte character that is found in the source string is mapped to the value %HFCFB.
- The definition of nondisplayable and invalid characters varies with the target mapping format. The IBM and Fujitsu character sets contain extensions that are not supported in the HP internal character set. When extension character codes are encountered, they are mapped to the nondisplayable character code for the HP character set.

  The most common definition of an invalid character code is a character pair that is expected to be a 2-byte code but has an invalid first or second byte.

  Any character mapped to either a nondisplayable or invalid character target code becomes nonrecoverable for conversion to the original format.

MBCS_FORMAT_CRT_FIELD_ Procedure

Summary

The MBCS_FORMAT_CRT_FIELD_ procedure formats Kanji only or mixed data types for specific terminal types.

Syntax for C Programmers

Note. In the TNS/E environment the CEXTDECS file uses the *int* data type for 32-bit values. This is a change from the TNS and TNS/R environments where CEXTDECS uses the *long* data type for 32-bit values.

```
#include <cextdecs(MBCS_FORMAT_CRT_FIELD_)>

short MBCS_FORMAT_CRT_FIELD_ ( __int32_t *source-string
                              ,__int32_t *destination-string
                              ,short source-length
                              ,short maximum-length
                              ,short intermediate
                              ,short terminal-type
                              ,short last-column
                              ,short *max-data-size
                              ,short *screen-start-col
                              ,[ char *shift-to-MBCS ]
                              ,[ char *shift-to-one-byte ]
                              ,[ short startmode ] );
```

Syntax for TAL Programmers

```
error-code := MBCS_FORMAT_CRT_FIELD_
                            ( source-string             ! i,o
                             ,destination-string        ! i,o
                             ,source-length             ! i
                             ,maximum-length            ! i
                             ,intermediate              ! i
                             ,terminal-type             ! i
                             ,last-column,              ! i
                             ,max-data-size             ! i,o
                             ,screen-start-col          ! i,o
                             ,[ shift-to-MBCS ]         ! i
                             ,[ shift-to-one-byte ]     ! i
                             ,[ startmode ] );          ! i
```

Parameters

error-code returned value

INT

returns a procedure error code.

 0 Successful completion of the translation
-1 Source string translation incomplete, ran out of destination buffer area or ran out of space in the screen field
-2 Unknown terminal type specified
-3 Invalid source string length
-4 Invalid character in Kanji-only source string
-5 Control string parameter too long
29 Required parameter missing

source-string input, output

INT(32) .EXT:ref:1

is the address of an extended address pointer to the source text string to be formatted. Upon return from the format function, this pointer points to the first unformatted byte in the source string. When the entire source string has been formatted, this pointer points to the position beyond the last byte of the source string.

destination-string input, output

INT(32) .EXT:ref:1

is the address of an extended address pointer to the location to receive the formatted text string. Upon return from the format function, this pointer points to the first unfilled byte in the destination string.

source-length input

INT:value

is the length, in bytes, of the source text string.

maximum-length input

INT: value

is the maximum allowable number of bytes of space usable in the output destination string.

intermediate input

INT:value

is a logical flag indicating the desired format of the translated data string.

When TRUE, the translated destination CRT field is in pre-EBCDIC format. This is an intermediate form which yields the final EBCDIC data format after passing the HP standard ASCII/EBCDIC translation routine. This is the normal form to use for formatting CRT fields which are then displayed through the HP SNAX access method.

When FALSE, the translated destination CRT field is in final EBCDIC format upon completion of this procedure.

terminal-type input

INT:value

indicates the terminal type to receive the formatted data.

4 Fujitsu F-6650/F-6680 with lowercase alphabet
5 Fujitsu F-6650/F-6680 with 1-byte Katakana
11 IBM character attribute device with lowercase alphabet
12 IBM character attribute device with 1-byte Katakana

last-column input

INT:value

indicates the width in columns of the display device.

max-data-size input, output

INT .EXT:ref:1

on input, contains an integer indicating the maximum number of displayable characters that can be held by the destination screen field. This value does not include the attribute byte.

on output, contains the byte count (length) of the translated data in the destination CRT field.

screen-start-col input, output

INT .EXT:ref:1

on input, contains the starting column on the current line of the screen where the first displayable data character may appear.

on output, contains an integer which represents the sum of the *screen-start-col* and the displayable length of the data inserted into the field by the translate function.

shift-to-MBCS input

STRING .EXT:ref:*

is an optional pointer to a string containing the escape, control, or other string to be used to indicate a shift to a multibyte character set in the destination text string. This string must always be in HP internal format (not EBCDIC), regardless of the final form of the destination string. When not supplied, a default string is used by the format routine (for Fujitsu terminals, it is [%H88, null]).

shift-to-one-byte input

STRING .EXT:ref:*

is an optional pointer to a string containing the escape, control, or other string to be used to indicate a shift to a 1-byte character set in the destination text string. This string must always be in HP internal format (not EBCDIC), regardless of the final form of the destination string. When not supplied, a default string is used by the format routine (for Fujitsu terminals, it is [%H89, null]).

startmode input

INT:value

specifies the optional start mode of the formatting operation. When this parameter has a value of 1, the formatting operation begins in the 1-byte mode. When this parameter has a value of 2, the formatting operation begins in the 2-byte (MBCS) mode. If any other value is specified, or if this parameter is not used, the starting mode is determined from the source string content. This parameter can be used to force a start in MBCS mode for a string that begins with double spaces. This is the only known ambiguous condition that might require the use of this parameter.

Consideration

Determining the size of the destination buffer is the responsibility of the calling procedure.

MBCS_FORMAT_ITI_BUFFER_ Procedure

Summary

The MBCS_FORMAT_ITI_BUFFER_ procedure formats ITI buffers for specific terminal types. This procedure is designed to support a known subset of double-byte character set SNA3270 display devices. Formatting is confined to the data area of the ITI buffer.

Syntax for C Programmers

Note. In the TNS/E environment the CEXTDECS file uses the *int* data type for 32-bit values. This is a change from the TNS and TNS/R environments where CEXTDECS uses the *long* data type for 32-bit values.

```
#include <cextdecs(MBCS_FORMAT_ITI_BUFFER_)>

short MBCS_FORMAT_ITI_BUFFER_ ( __int32_t *source-string
                               ,__int32_t *destination-string
                               ,short source-length
                               ,short maximum-length
                               ,short intermediate
                               ,short terminal-type
                               ,short maximum-col-count
                               ,short *finished-length
                               ,short *screen-col-count
                               ,[ char *shift-to-MBCS ]
                               ,[ char *shift-to-one-byte ]
                               ,[ short startmode ] );
```

Syntax for TAL Programmers

```
error-code := MBCS_FORMAT_ITI_BUFFER_
                            ( source-string              ! i,o
                             ,destination-string         ! i,o
                             ,source-length              ! i
                             ,maximum-length             ! i
                             ,intermediate               ! i
                             ,terminal-type              ! i
                             ,maximum-col-count          ! i
                             ,finished-length            ! o
                             ,screen-col-count           ! o
                             ,[ shift-to-MBCS ]          ! i
                             ,[ shift-to-one-byte ]      ! i
                             ,[ startmode ] );           ! i
```

Parameters

error-code returned value

INT

returns a procedure error code.

0	Successful completion of the translation
-1	Source string translation incomplete, ran out of destination buffer area
-2	Unknown terminal type specified
-3	Invalid source string length
-4	Invalid character in Kanji only source string
-5	Control string parameter too long
29	Required parameter missing

source-string input, output

INT(32) .EXT:ref:1

is the address of an extended pointer to the source text string to be formatted. Upon return from the format function, this pointer is advanced to point to the byte following the last byte in the source string that was successfully processed by the format ITI buffer operation.

destination-string input, output

INT(32) .EXT:ref:1

is the address an extended pointer to the location to receive the formatted text string. Upon return from the format function, this pointer is advanced to point to the byte following the last byte in the destination string that was filled by the format ITI buffer operation.

source-length input

INT:value

is the length, in bytes, of the source text string.

maximum-length input

INT:value

is the maximum allowable number of bytes of space usable in the output destination string.

intermediate input

INT:value

is a logical flag indicating the desired format of the translated data string.

When TRUE, the translated destination text string is in an intermediate form which yields the final EBCDIC data format after passing the HP standard ASCII/EBCDIC translation routine. This is the normal form to use for formatting buffers which are then displayed through the HP SNAX access method.

When FALSE, the translated destination text string is in final EBCDIC format upon completion of this procedure.

terminal-type input

INT:value

indicates the terminal type to receive the formatted data.

0 IBM 3274-series
1 IBM Emulation on IBM5550 with lowercase alphabet
2 IBM Emulation on IBM5550 with 1-byte Katakana
4 Fujitsu F-6650/F-6680 with lowercase alphabet
5 Fujitsu F-6650/F-6680 with 1-byte Katakana
11 IBM character attribute device with lowercase alphabet
12 IBM character attribute device with 1-byte Katakana

maximum-col-count input

INT:value

indicates the width (in columns) of the display device.

finished-length output

INT .EXT:ref:1

contains the byte count of the part of the destination string containing successfully translated data.

screen-col-count output

INT .EXT:ref:1

contains the displayable column count of the formatted buffer.

shift-to-MBCS input

STRING .EXT:ref:*

is an optional pointer to a string containing the escape, control, or other string used to indicate a shift to a multibyte character set in the destination text string. This string must always be in HP internal format (not EBCDIC), regardless of the final form of the destination string. When not supplied, a default string is used by the format routine:

for IBM terminals	[%H0E, null]
for Fujitsu terminals	[%H88, null]
for IBM terminals with character attribute device	[null, %H03, %H88, %HA2, %H38]

shift-to-one-byte input

STRING .EXT:ref:*

is an optional pointer to a string containing the escape, control, or other string used to indicate a shift to a 1-byte character set in the destination text string. This string must always be in HP internal format (not EBCDIC), regardless of the final form of

the destination string. When not supplied, a default string is used by the format routine:

for IBM terminals	[%H0F, null]
for Fujitsu terminals	[%H89, null]
for IBM terminals with character attribute device	[null, %H03, %H88, %HA2, %H00]

startmode input

INT:value

specifies the optional start mode of the formatting operation. When this parameter has a value of 1, the formatting operation begins in the 1-byte mode. When this parameter has a value of 2, the formatting operation begins in the 2-byte (MBCS) mode. If any other value is specified, or if this parameter is not used, the starting mode is determined from the source string content. This parameter can be used to force a start in MBCS mode for a string that begins with Kanji double-spaces. This is the only known ambiguous condition that might require the use of this parameter.

Consideration

Determining the size of the destination buffer is the responsibility of the calling procedure.

MBCS_MB_TO_SB_ Procedure

Summary

This procedure and the companion MBCS_SB_TO_MB_ procedure allow conversion of the ninety-four displayable characters of the ASCII character set between 1-byte ASCII and characters of the specified MBCS. The MBCS_MB_TO_SB_ procedure converts multibyte characters to the corresponding 1-byte ASCII characters.

Syntax for C Programmers

```
#include <cextdecs(MBCS_MB_TO_SB_)>

void MBCS_MB_TO_SB_ ( char *mbytestring
                     ,short mbytecount
                     ,char *sbytestring
                     ,short sbytecount
                     ,short *rbytecount
                     ,[ short charset ]
                     ,[ short *charinfo ] );
```

Syntax for TAL Programmers

```
CALL MBCS_MB_TO_SB_  ( mbytestring:mbytecount    ! i:i
                      ,sbytestring:sbytecount    ! o:i
                      ,rbytecount                ! o
                      ,[ charset ]               ! i
                      ,[ charinfo ] );           ! o
```

Parameters

mbytestring:*mbytecount* input:input

STRING .EXT:*, INT:value

specifies a text string with ASCII-equivalent characters in the multibyte character set identified by *charset*. *mbytestring* is the input text string to be converted by this procedure; it must be exactly *mbytecount* bytes long.

sbytestring:*sbytecount* output:input

STRING .EXT:*, INT:value

returns the converted text string containing 1-byte characters. The string variable *sbytestring* must be exactly *sbytecount* bytes long.

rbytecount output

INT .EXT:ref:1

returns the actual byte length of the converted text string contained in *sbytestring*.

charset input

INT:value

is the optional identifier of the reference MBCS. When omitted or null, the default MBCS character set identifier from the MBCS_DEFAULTCHARSET_ procedure is used. These multibyte character sets are supported by this procedure:

1 HP Kanji
9 HP Hangul
10 HP Chinese Big 5
11 HP Chinese PC
12 HP KSC5601

charinfo output

INT .EXT:ref:1

is an optional parameter that returns an indication of any cause of failure. Except for file-system error 29 (required missing parameter), this procedure does not initiate error indications but simply passes on the errors returned by the MBCS_CHAR_ procedure. For returned values and interpretations, see MBCS_CHAR_ Procedure. Upon return of an error from the MBCS_CHAR_ procedure, the operation is aborted and processing is returned to the caller.

Considerations

The input text string may contain any combination of mixed 1-byte and multibyte characters. Eligible multibyte characters are converted to the appropriate ASCII equivalent characters. Other characters or bytes encountered in the input string are moved to the output string without change. Multibyte blanks are converted to equivalent sized strings of ASCII blanks (%H20).

MBCS_REPLACEBLANK_ Procedure

Summary

The MBCS_REPLACEBLANK_ procedure replaces nonstandard blanks.

Within multibyte character sets, there are usually characters defined that have a blank display attribute of the same width as the other multibyte characters of the same character set. Since these characters do not have an internal representation that is recognized by HP subsystems as blanks, they cannot be used as word separators or delimiters in the same manner as the 1-byte space character.

HP recommends that multibyte strings of blanks be used instead of normally defined multibyte character blanks. It is not possible to guarantee that this recommendation will always be followed, so it is also recommended that data be processed to replace the multibyte blanks of the current character set with blank strings of equivalent length.

This procedure allows callers to replace the normally-defined multibyte blank characters with equivalent-sized strings of blank (%H20) characters.

Syntax for C Programmers

```
#include <cextdecs(MBCS_REPLACEBLANK_)>

void MBCS_REPLACEBLANK_ ( char *bytestring
                         ,short bytecount
                         ,[ short charset ]
                         ,[ short *charinfo ] );
```

Sytax for TAL Programmers

```
CALL MBCS_REPLACEBLANK_   ( bytestring            ! i
                           ,bytecount             ! i
                           ,[ charset ]           ! i
                           ,[ charinfo ] );       ! i,o
```

Parameters

bytestring input

STRING .EXT:ref:*

is a pointer to a buffer containing a properly formed text string that may contain any mixture of 1-byte and MBCS characters. A properly formed text string may not begin with the second or subsequent byte of an MBCS character or end with any byte of a MBCS character other than the last. (See MBCS_TRIMFRAGMENT_ Procedure.) The contents of the *bytestring* pointer are not altered by the shift operation.

bytecount input

INT:value

is an integer variable containing the length in bytes of the text string *bytestring*.

charset input

INT:value

identifies the multibyte character set (MBCS) to be used. If *charset* is omitted or null, the default MBCS from the MBCS_DEFAULTCHARSET_ procedure is used. MBCS_REPLACEBLANK_ does not examine or validate the character set identification but simply passes it on to the MBCS_CHAR_ procedure. All MBCSs supported by the MBCS_CHAR_ procedure are inherently supported by this

procedure. This procedure replaces different multibyte character pairs depending on the *charset* value.

charset	Character	Becomes
HP Kanji	%H8140	%H2020
HP Hangul	%HFC20	%H2020
HP Chinese Big 5	%HA140	%H2020
HP Chinese PC	%H8140	%H2020
HP KSC5601	%HA1A1	%H2020

charinfo input, output

INT .EXT:ref:1

indicates the cause of failure of the requested test. This procedure returns file-system error 29 to indicate missing required parameters; other error indications are passed back from the MBCS_CHAR_ procedure. For returned values and interpretations, see MBCS_CHAR_ Procedure. Upon return of an error from the MBCS_CHAR_ procedure, the operation is aborted and processing is returned to the caller.

Considerations

Except for the HP Hangul data format, none of the supported internal MBCS data formats use character definitions that include single-byte values of %H20 in combination with other values to form multibyte characters. In other words, with the exception of HP Hangul, none of the characters of the supported internal multibyte character sets contain embedded blanks. When using any of the supported MBCSs other than HP Hangul, following the use of this procedure, any remaining bytes having the value of %H20 can be interpreted as ordinary blanks.

When the HP Hangul character set is specified, extra care is required because byte values of %H20 can appear as 1-byte blanks, the second byte of multibyte characters, or in pairs as 2-byte blanks.

Related Programming Manual

For programming information about the MBCS_REPLACEBLANK_ procedure, see the *Guardian Programmer's Guide*.

MBCS_SB_TO_MB_ Procedure

Summary

This procedure and the companion MBCS_MB_TO_SB_ procedure are provided to allow conversion of the ninety-four displayable characters of the ASCII character set between 1-byte ASCII and characters of the specified MBCS. The MBCS_SB_TO_MB_ procedure converts 1-byte ASCII characters to the corresponding multibyte characters.

Syntax for C Programmers

```
#include <cextdecs(MBCS_SB_TO_MB_)>

void MBCS_SB_TO_MB_ ( char *sbytestring
                     ,short sbytecount
                     ,char *mbytestring
                     ,short mbytecount
                     ,short *rbytecount
                     ,[ short charset ]
                     ,[ short *charinfo ] );
```

Syntax for TAL Programmers

```
CALL MBCS_SB_TO_MB_ ( sbytestring:sbytecount      ! i:i
                     ,mbytestring:mbytecount      ! o:i
                     ,rbytecount                  ! o
                     ,[ charset ]                 ! i
                     ,[ charinfo ] );             ! o
```

Parameters

sbytestring:*sbytecount* input:input

STRING .EXT:*, INT:value

specifies a text string with 1-byte ASCII characters. *sbytestring* is the input text string to be converted by this procedure; it must be exactly *sbytecount* bytes long.

mbytestring:*mbytecount* output:input

STRING .EXT:*, INT:value

returns the converted text string containing multibyte characters. The string variable *mbytestring* must be exactly *mbytecount* bytes long.

rbytecount output

INT .EXT:ref:1

returns the actual byte length of the converted text string contained in *mbytestring*.

charset input

INT:value

is the optional identifier of the reference MBCS. When omitted or null, the default MBCS character set identifier from the MBCS_DEFAULTCHARSET_ procedure is used. These multibyte character sets are supported by this procedure:

1 HP Kanji
9 HP Hangul
10 HP Chinese Big 5
11 HP Chinese PC
12 HP KSC5601

charinfo output

INT .EXT:ref:1

is an optional parameter that returns an indication of any cause of failure. is an optional reference parameter used to return an indication of any cause of failure of the requested test. Except for file-system error 29 (missing parameter), this procedure does not initiate error indications but simply passes on the errors returned by the MBCS_CHAR_ procedure. For returned values and interpretations, see MBCS_CHAR_ Procedure. Upon return of an error from the MBCS_CHAR_ procedure, the operation is aborted and processing is returned to the caller.

Considerations

The input text string may contain any combination of mixed 1-byte and multibyte characters. All 1-byte ASCII characters are converted to the appropriate ASCII equivalent characters in the specified multibyte character set. Other characters or bytes encountered in the input string are moved to the output string without change. All 1-byte blanks are moved without conversion or extension.

MBCS_SHIFTSTRING_ Procedure

Summary

The MBCS_SHIFTSTRING_ procedure upshifts or downshifts all alphabetic characters in a multibyte character set (MBCS) string. Both multibyte alphabetic characters of the specified MBCS and 1-byte alphabetic characters of the ASCII character set are case shifted by this procedure.

Syntax for C Programmers

```
#include <cextdecs(MBCS_SHIFTSTRING_)>

void MBCS_SHIFTSTRING_ ( char *bytestring
                        ,short bytecount
                        ,short casebit
                        ,[ short charset ]
                        ,[ short *charinfo ] );
```

Syntax for TAL Programmers

```
CALL MBCS_SHIFTSTRING_ ( bytestring              ! i
                        ,bytecount               ! i
                        ,casebit                 ! i
                        ,[ charset ]             ! i
                        ,[ charinfo ] );         ! o
```

Parameters

bytestring input

STRING .EXT:ref:*

is a pointer to a buffer containing a properly formed text string that may contain any mixture of 1-byte and MBCS characters. A properly formed text string may not begin with the second or subsequent byte of an MBCS character or end with any byte of an MBCS character other than the last. (See the MBCS_TRIMFRAGMENT_ Procedure.) The contents of the *bytestring* pointer are not altered by the shift operation.

bytecount input

INT:value

is an integer variable containing the length in bytes of the text string *bytestring*.

casebit input

INT:value

is a variable indicating the type of case shift to be applied to the 1-byte alphabetic characters in the bytestring. When the case bit (bit <15>) is set to 0, an upshift is requested; when the case bit is set to 1, a downshift is requested.

charset input

INT:value

identifies the multibyte character set (MBCS) to be used. If *charset* is omitted or null, the default MBCS from the MBCS_DEFAULTCHARSET_ procedure is used. This procedure does not examine or validate the character set identification but simply passes it on to the MBCS_CHAR_ procedure. All MBCSs supported by the MBCS_CHAR_ procedure are inherently supported by this procedure.

charinfo output

INT .EXT:ref:1

indicates the cause of failure of the requested test. This procedure returns file-system error 29 to indicate missing required parameters; other error indications are returned by the MBCS_CHAR_ procedure. (For returned values and interpretations, see MBCS_CHAR_ Procedure.) Upon return of an error from the MBCS_CHAR_ procedure, the shift operation is aborted and processing is returned to the caller.

Consideration

You must not use the SHIFTSTRING procedure with text that contains multibyte characters because of the potential damage to such characters. To avoid such potential damage, the SHIFTSTRING procedure is replaced by the MBCS_SHIFTSTRING_ procedure whenever an MBCS is installed.

The MBCS_SHIFTSTRING_ procedure appears similar to the SHIFTSTRING procedure but has significant differences: the characteristics of parameters are different and two parameters are added. This procedure handles upshift and downshift operations on a string of mixed single and multibyte characters—a function beyond the capability of the current case-shifting procedures.

Related Programming Manual

For programming information about the MBCS_SHIFTSTRING_ procedure, see the *Guardian Programmer's Guide*.

MBCS_TANDEM_TO_EXTERNAL_ Procedure

Summary

The MBCS_TANDEM_TO_EXTERNAL_ procedure translates a text string from HP internal format to a specified external text format.

Syntax for C Programmers

Note: In the TNS/E environment the CEXTDECS file uses the *int* data type for 32-bit values. This is a change from the TNS and TNS/R environments where CEXTDECS uses the *long* data type for 32-bit values.

```
#include <cextdecs(MBCS_TANDEM_TO_EXTERNAL_)>

short MBCS_TANDEM_TO_EXTERNAL_ ( __int32_t *source-string
                                ,__int32_t *destination-string
                                 ,short source-length
                                 ,short maximum-length
                                 ,short intermediate
                                 ,short external-form
                                 ,short *finished-length
                                 ,[ char *shift-to-MBCS ]
                                ,[ char *shift-to-one-byte ] );
```

Syntax for TAL Programmers

```
error-code := MBCS_TANDEM_TO_EXTERNAL_
                            ( source-string              ! i,o
                             ,destination-string         ! i,o
                             ,source-length              ! i
                             ,maximum-length             ! i,o
                             ,intermediate               ! i
                             ,external-form              ! i
                             ,finished-length            ! o
                             ,[ shift-to-MBCS ]          ! i
                             ,[ shift-to-one-byte ] );   ! i
```

Parameters

error-code returned value

INT

returns a procedure error code. Possible error codes are:

0 Successful completion of the translation
-1 Translation truncated due to lack of destination buffer space
-2 Unknown translation requested
-3 Invalid source string length
-4 Invalid character in Kanji-only source string
-5 Control string parameter too long
29 Required parameter missing

source-string input, output

INT(32) .EXT:ref:1

is a pointer to a double-word integer containing the extended address of the source text string to be translated. After translation, this address points to the byte following the last byte in the source string that was successfully translated.

destination-string input, output

INT(32) .EXT:ref:1

is a pointer to a double-word integer containing the extended address of the location to receive the translated text string. After translation, this address points to the byte following the last byte in the destination string used by the translation operation.

source-length input

INT:value

is the length, in bytes, of the source text string.

maximum-length input, output

INT .EXT:ref:1

on input, is the maximum allowable number of bytes of space in the output destination string.

While all the formal parameters (*shift-to-MBCS* and *shift-to-one-byte*) are mandatory for a string translation, specifying only the *source-length*, *maximum-length*, and *external-form* parameters (omitting all other parameters), returns the maximum length required for a destination string, without any string translation.

intermediate input

INT:value

is an optional logical flag indicating the desired format of the translated data string.

For translations to an EBCDIC type of data format, this parameter is interpreted as follows:

When TRUE, the translated destination text string is in an intermediate form which yields the final EBCDIC data format after passing the HP standard ASCII/EBCDIC translation routine.

When FALSE, the translated destination text string is in final EBCDIC format upon completion of this procedure.

For translations to a format containing JIS standard Kanji and a JIS or ASCII-like 1-byte character set, this parameter is interpreted as follows:

When TRUE, the translated data stream is in shift-in/ shift-out (SI/SO) format. An initial ASCII SI or SO character is placed at the beginning of the destination string, depending on the value of the first byte of translated text. ASCII SI/SO characters frame data character sub-strings which represent byte values of octal 240 or greater. The high-order bit of each byte in these sub-strings is set off.

When FALSE, the translated data stream is in an eight-bit data format. SI/SO characters are not inserted in the translated data text stream.

external-form input

INT:value

indicates the target format for the destination text translation. The four high-order bits of this parameter (*external-form.<0:3>*) are reserved for use by the Kanji

EM3270 product. For all other uses, the four high order bits must be set to null. The twelve low-order bits of this parameter (*external-form.<4:15>*) have this meaning:

- IBM external formats
  - Data stream without substring frames

    0 IBM Kanji only
  - Data stream using SO/SI substring frames

    1 IBM Kanji EBCDIC

    2 IBM Kanji/Katakana-EBCDIC
  - Data stream using character attribute substring framing (IBM 3270 data stream only)

    11 IBM Kanji EBCDIC

    12 IBM Kanji/Katakana-EBCDIC
- JEF external formats
  - Data stream using KI/KO substring frames

    3 JEF (Fujitsu) Kanji only

    4 JEF (Fujitsu) Kanji EBCDIC

    5 JEF (Fujitsu) Kanji/Katakana-EBCDIC
- Other external formats

    8 JIS X0208 Kanji/JIS X0201 (was C6226/C6220)

finished-length output

INT .EXT:ref:1

contains the byte count of the part of the destination string containing data which was successfully translated. When this parameter is missing, the translate function attempts to return an estimate of the space required for the destination string.

shift-to-MBCS input

STRING .EXT:ref:*

is an optional pointer to a string containing the escape or control string used to indicate a shift to a multibyte character set in the destination text string. This string must always be in HP internal (not EBCDIC) character format, regardless of the final form of the source string.

This procedure does not contain logic for identifying escape or control strings. The control strings that are used are either the specified default control strings or user-supplied alternative control strings. In earlier versions of this procedure, control strings were null delimited. While null-delimited control strings are still supported,

an alternative format that supports control strings containing null bytes is now also provided. The IBM Kanji character attribute (described following) is an example of an alternative format.

The alternative format must be expressed as (null flag, count, string), where the first byte is a null flag indicator of the alternative string form, the second byte contains an integer value representing the length of the control string, and the third and subsequent bytes represent the value of the control string.

The minimum length for a control string is 1 byte, and the maximum length is 20 bytes. If the procedure receives a zero-length control string, the results are undefined.

When the control string values are not user-supplied, the default values used for control strings expressed in the original null-delimited format are as follows:

IBM Kanji	[%H0E,null]
JEF (Fujitsu) Kanji	[%H88,null]
JIS X0208 Kanji	[%H1B, %H24,%H42,null]

When the control string values are not user-supplied, the default values used for control strings expressed in the alternative format are as follows:

IBM Kanji character attribute	[null, %H03, %H88, %HA2, %H38]

shift-to-one-byte input

STRING .EXT:ref:*

is an optional pointer to a string containing the escape or control string used to indicate a shift to a 1-byte character set in the source text string. This string must always be in HP internal (not EBCDIC) character format, regardless of the final form of the source string.

This procedure does not contain logic for identifying escape or control strings. The control strings that are used are either the specified default control strings or user-supplied alternative control strings. In earlier versions of this procedure, control strings were null delimited. While null-delimited control strings are still supported, an alternative format that supports control strings containing null bytes is now also provided. The IBM Kanji character attribute (described following) is an example of an alternative format.

The alternative format that supports user-supplied control strings containing null bytes must be expressed as (null flag, count, string), where the first byte is a null flag indicator of the alternative string form, the second byte contains an integer value representing the length of the control string, and the third and subsequent bytes represent the value of the control string.

The minimum length for a control string is 1 byte, and the maximum length is 20 bytes. If the procedure receives a zero-length control string, the results are undefined.

When the control string values are not user-supplied, the default values used for control strings expressed in the original null-delimited format are as follows:

IBM Kanji	[%H0F,null]
JEF (Fujitsu) Kanji	[%H89,null]
JIS X0208 Kanji	[%H18,%H28,%H4A,null]

When the control string values are not user-supplied, the default values used for control strings expressed in the alternative format is as follows:

IBM Kanji	
character attribute	[null, %H03, %H88, %HA2, %H00]

Considerations

- All parameters except *shift-to-MBCS* and *shift-to-one-byte* are necessary for a string translation operation.
- In general, translations to external text formats yield text strings of increased length. If less than this amount of space is allowed, the translation procedure might fail due to insufficient space in the destination string. To determine the maximum length of the destination string after translation, specify the *source-length*, *maximum-length*, and *external-form* parameters, and the other formal parameters can be omitted.
- In the HP internal character sets, text bytes having the byte value x20 are used to represent blank characters or spaces in both the 1-byte and the 2-byte character sets. A 1-byte blank is represented by a single x20 byte. A 2-byte blank is represented by two consecutive x20 bytes. When converting text from the HP internal format to an external format, some ambiguity might be introduced in choosing the 1-byte or 2-byte mode for the external form of two or more consecutive x20 bytes found in the internal text string.

  For conversion of Kanji-only text, any 1-byte character (including a 1-byte blank) is considered invalid and causes an error to be returned from this procedure. Two consecutive 1-byte blanks in Kanji-only text is converted to a 2-byte blank in the external format.

  In mixed-text conversion operations, when blank bytes are encountered, the conversion logic avoids changing the 1-byte/2-byte mode if possible. If compatible, the external 1-byte/2-byte mode of the blanks is assumed to be the same as that of the context before the location of the blank bytes. One or more blanks found at the start of a field, or following other 1-byte characters, are treated as 1-byte blanks. One or more pairs of blanks following other 2-byte characters are treated as 2-byte

blanks. This table summarizes the blank-handling logic followed in text conversion operations from HP internal to other external formats.

Target Field Type	Internal Text Number of Blanks	Location in Field	External Text Number and Type of Blanks
Kanji only	1	Any	none (invalid)
Kanji only	2	Any	one 2-byte
Mixed text	1	Any	one 1-byte
Mixed text	2	Beginning	two 1-byte
Mixed text	2	Following 1-byte	two 1-byte
Mixed text	2	Following 2-byte	one 2-byte

Note. The common representation for a 2-byte blank character in the Shift-JIS character code is x8140, While HP subsystem software might not recognize and treat this character code as a blank, if it is present in an internal text string, it will be mapped to a 2-byte blank when the text is converted to an external Kanji character set.

- When MBCS_TANDEM_TO_EXTERNAL_ finds invalid or nondisplayable two-byte characters in the source string, it maps them to reserved values as follows:

Destination Format	Invalid Pairs Map to	Nondisplayable Pairs Map to
IBM	%HFEFE	%HFEFD
Fujitsu	%HA0FE	%HA0FD
JIS	%H2222	%H2223

- The definition of nondisplayable and invalid characters varies with the target mapping format.

  Mapping between HP and IBM formats is done with mapping tables. There are many HP 2-byte character codes that do not have defined fonts. These character codes do not have defined character code targets in the IBM format, and thus they are mapped to the nondisplayable character code.

  Mapping between HP formats and Fujitsu or JIS formats is done by algorithm. The HP internal character set is larger than the supported Fujitsu or JIS character set. Valid 2-byte character codes from the HP internal character set are mapped to the target nondisplayable character code.

- The most common definition of an invalid character code is a character pair that is expected to be a 2-byte code but has an invalid first or second byte.

  Any character mapped to either a nondisplayable or invalid character target code becomes nonrecoverable for conversion to the original format.

MBCS_TESTBYTE_ Procedure

Summary

The MBCS_TESTBYTE_ procedure returns the identification of a specified byte contained within a text string of mixed data.

Syntax for C Programmers

```
#include <cextdecs(MBCS_TESTBYTE_)>

short MBCS_TESTBYTE_ ( char *buffer
                      ,short bytecount
                      ,short *testindex
                      ,[ short charset ]
                      ,[ short *charinfo ] );
```

Syntax for TAL Programmers

```
result := MBCS_TESTBYTE_ ( buffer                ! i
                          ,bytecount             ! i
                          ,testindex             ! i,o
                          ,[ charset ]           ! i
                          ,[ charinfo ]);        ! o
```

Parameters

result returned value

INT

returns the identification of the byte contained in *buffer*[*testindex*].

0 1-byte character
1 First byte of a multibyte character
2 Intermediate byte of a multibyte character
3 Last byte of a multibyte character

buffer input

STRING .EXT:ref:*

is a pointer to a buffer containing a properly formed text string which may contain any mixture of 1-byte and multibyte characters. A properly formed text string may not begin with the second or subsequent byte of a multibyte character or end with a fragment of a multibyte character. The *buffer* is not altered by the test operation. This procedure does not alter the contents of the text string referenced by *buffer*.

bytecount input

INT:value

is the number of bytes in the *buffer* text string.

testindex input, output

INT .EXT:ref:1

on input, specifies the string index of the byte in *buffer* to be tested.

On output, if *result* indicates that *buffer*[*testindex*] is part of a multibyte character, then *testindex* contains the byte index of the first byte of the multibyte character containing the tested byte.

charset input

INT:value

identifies the multibyte character set (MBCS) to be used. If *charset* is omitted or null, the MBCS returned from the MBCS_DEFAULTCHARSET_ procedure is used. This procedure does not examine or validate the character set identification, but simply passes it on to the MBCS_CHAR_ procedure. All MBCSs supported by the MBCS_CHAR_ procedure are supported by this procedure.

charinfo output

INT .EXT:ref:1

indicates the cause of failure of the requested test. This procedure returns file-system error 29 to indicate missing required parameters; other error indications are passed back from the MBCS_CHAR_ procedure. For returned values and interpretations, see MBCS_CHAR_ Procedure. Upon return of an error from the MBCS_CHAR_ procedure, the operation is aborted and processing is returned to the caller.

Considerations

- A simple range check is not adequate to establish positive identification of MBCS byte usages.

  The set of second-byte values used by Shift-JIS Kanji characters overlaps the byte value ranges of ASCII, 1-byte Katakana, and the set of byte values used for the first byte of Shift-JIS Kanji characters. Taken out of context, the second byte of a Shift-JIS character could be mistaken as an ASCII character, a 1-byte Katakana character, or the first byte of a Kanji character.

  A combination of relative position and value range analysis is required to correctly establish usage identity as a 1-byte character or as a particular MBCS byte identity. This is the basic purpose of the MBCS_TESTBYTE_ procedure.

- To obtain proper results from the use of this procedure, the caller must ensure that the string referenced by the *buffer* parameter is a properly formed text string. A properly formed text string meets these criteria:

  - The first byte (*buffer*[0]) is either a 1-byte character or the first byte of an MBCS character. It can be assumed that a displayable line of text input from a terminal meets this requirement. Do not assume that an arbitrarily selected extract from a text string meets this requirement.

  - The last byte in a properly formed text string is either a 1-byte character or an MBCS final byte. A line of text typed at a terminal in conversational mode does not necessarily meet this requirement. (MBCS_TRIMFRAGMENT_ Procedure.) Do not assume that an arbitrarily selected extract from a text string meets this requirement.

- This procedure does not alter the contents of the text string.

- The MBCS_TESTBYTE_ procedure tests isolated bytes in a text string for MBCS characteristics. On each call to this procedure, the *buffer* text string is analyzed from the beginning up to a point just past *buffer*[*testindex*].

  Repetitive analysis of a text string from the beginning is not particularly efficient. This procedure is not the function of choice for iterative operations where each byte or character in a text string is to be tested and processed. For operations where it is necessary to test and process each byte in a text string, greater efficiency can be achieved by using a user-coded procedure which progressively tests and processes as it works its way through the target text string. For a sample user-coded procedure, see MBCS_CHAR_ Procedure.

Related Programming Manual

For programming information about the MBCS_TESTBYTE_ procedure, see the *Guardian Programmer's Guide*.

MBCS_TRIMFRAGMENT_ Procedure

Summary

The MBCS_TRIMFRAGMENT_ procedure detects and trims trailing multibyte character fragments from text strings.

In conversational mode operations, a string read from a terminal might contain a partial multibyte character at the end of the string; the result perhaps of a read operation of an odd length. If the terminal operator attempts to enter a string of greater length than the requested read count, the read operation will complete upon reaching the requested read count. If the characters entered by the terminal operator were all 2-byte characters, then the input byte which satisfies the read count becomes the first byte of a 2-byte character. The second byte of the character is lost. Since multibyte characters and 1-byte characters may be freely mixed in text strings, with a multibyte character beginning at any byte location, a trailing fragment can occur at the end of any conversational mode read operation. The multibyte character fragment is an undesirable effect of the use of 1-byte I/O operations to handle multibyte characters.

Syntax for C Programmers

```
#include <cextdecs(MBCS_TRIMFRAGMENT_)>

void MBCS_TRIMFRAGMENT_ ( char *bytestring
                         ,short *bytecount
                         ,[ short charset ]
                         ,[ short *charinfo ] );
```

Syntax for TAL Programmers

```
CALL MBCS_TRIMFRAGMENT_   ( bytestring         ! i
                          ,bytecount           ! i,o
                          ,[ charset ]         ! i
                          ,[ charinfo ] );     ! o
```

Parameters

bytestring input

STRING .EXT:ref:*

is a pointer to a buffer containing a text string which may contain any mixture of 1-byte and MBCS characters. The content of the *bytestring* pointer is not altered by the trim operation.

bytecount input, output

INT .EXT:ref:1

on input, is an integer variable containing the length in bytes of the text string *bytestring*.

on output, is an integer variable containing the length in bytes of the text string *bytestring*. The output value will be less than the input value when a multibyte character fragment has been found and trimmed from the text string.

charset input

INT:value

identifies the multibyte character set (MBCS) to be used. If *charset* is omitted or null, the MBCS returned from the MBCS_DEFAULTCHARSET_ procedure is used. This procedure does not examine or validate the character set identification, but simply passes it on to the MBCS_CHAR_ procedure. All MBCSs supported by the MBCS_CHAR_ procedure are supported by this procedure.

charinfo output

INT .EXT:ref:1

indicates the cause of failure of the requested test. This procedure may return an indication that the specified character set is not recognized (-2), or of missing required parameters (file-system error 29).

Related Programming Manual

For programming information about the MBCS_TRIMFRAGMENT_ procedure, see the *Guardian Programmer's Guide*.

MESSAGESTATUS Procedure

Summary

MESSAGESTATUS determines if a particular message received through READUPDATE has been canceled.

Syntax for C Programmers

```
#include <cextdecs(MESSAGESTATUS)>

short MESSAGESTATUS ( [ short message-tag ] );
```

Syntax for TAL Programmers

```
status := MESSAGESTATUS ( [ message-tag ] );  ! i
```

Parameters

status returned value

INT

is a value indicating the cancellation status of the indicated message:

1 Message has been canceled.
0 Message has not been canceled.
-1 No pending message is associated with the given tag (see “Considerations”).

message-tag input

INT:value

is the message tag returned from FILE_GETRECEIVEINFO_ or RECEIVEINFO following receipt of the message. If omitted, the most recently received message is indicated.

Considerations

- If a message is canceled, any information supplied to REPLY (which must still be called) is not passed back to the message originator. A message can be canceled because the originator called CANCEL, CANCELREQ, FILE_CLOSE_ , CLOSE, or

certain forms of AWAITIO; cancellation will also be caused by a stop of the originating process, failure of the originator's processor, or a network communication failure.

- This procedure is best used for a program that is concerned about one particular request. If a program deals with many requests concurrently and constantly monitors $RECEIVE, use of system message -38 (queued message cancellation) may be more appropriate. You can be notified when pending messages are canceled with a system message -38 if SETMODE 80 has been enabled (see SETMODE Procedure).

Related Programming Manual

For programming information about the MESSAGESTATUS procedure, see the *Guardian Programmer's Guide*.

MESSAGESYSTEMINFO Procedure

Summary

MESSAGESYSTEMINFO measures the current number of messages to or from a process so it can issue a warning when a limit is nearly reached. MESSAGESYSTEMINFO is used with CONTROLMESSAGESYSTEM.

You can use MESSAGESYSTEMINFO during checkout of a program that uses CONTROLMESSAGESYSTEM, for example, to verify that initialization was done as expected.

Syntax for C Programmers

```
#include <cextdecs(MESSAGESYSTEMINFO)>

short MESSAGESYSTEMINFO ( short itemcode
                        ,short *value );
```

Syntax for TAL Programmers

```
error := MESSAGESYSTEMINFO ( itemcode          ! i
                           ,value );           ! o
```

Parameters

error returned value

INT

returns these file-system error numbers:

0 Successful, no error
2 Bad *itemcode*
21 Bad *value*
22 Bounds error
29 Missing parameter

See the *Guardian Procedure Errors and Messages Manual* for more information.

itemcode input

INT:value

specifies the item code of the information to be retrieved. See this list.

value output

INT .EXT:ref:1

returns the value indicated in this list.

Item Code	Returned Value
0	Current limit on the number of messages to this process, as set by CONTROLMESSAGESYSTEM.
1	Current limit on the number of messages from this process, as set by CONTROLMESSAGESYSTEM.
4	The number of outstanding messages to this process.
5	The number of outstanding messages from this process.

Considerations

MESSAGESYSTEMINFO is inherently tied to the internal workings of the message system, so one or more of its functions might not be supported by future versions of the message system. If it returns a nonzero *error*, the caller should record the error, but should otherwise ignore the error.

MOM Procedure (Superseded by PROCESS_GETINFOLIST_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The MOM procedure provides a process with the 4-word process ID of its creator. The process ID is a 4-word array, where *process-id* [0:2] contains the process name or creation timestamp and *process-id* [3] contains the *cpu,pin* for the process.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
CALL MOM ( process-id );                    ! o
```

Parameters

process-id output

INT:ref:4

is the 4-word array where MOM returns the process ID of the caller's creator. For an unnamed process, *process-id* is:

[0].<0:1>	2
.<2:7>	Reserved
.<8:15>	System number (0 through 254)
[1:2]	Low-order 32 bits of creation timestamp
[3].<0:3>	Reserved
.<4:7>	processor number where the process is executing
.<8:15>	PIN assigned by the operating system to identify the process in the processor

For a named local process, *process-id* is:

[0:2]	*$process-name*
[3].<0:3>	Reserved
.<4:7>	processor number where the process is executing
.<8:15>	PIN assigned by the operating system to identify the process in the processor

For a named remote process, *process-id* is:

[0].<0:7>	"\" (ASCII backslash)
.<8:15>	System number
[1:2]	*$process-name*
[3].<0:3>	Reserved
.<4:7>	processor number where the process is executing
.<8:15>	PIN assigned by the operating system to identify the process in the processor

Considerations

- Calling MOM from a named process or process pair

  If the caller is a single named process (that is, the caller is the primary process of a named process pair with no backup process), zeros are returned in *process-id*.

  If the caller of MOM is the primary process of a named process pair and there is a backup process, the process ID of the backup is returned.

  If the caller of MOM is the backup process of a named process pair, the process ID of the primary is returned.

- Passing the process ID to the system procedures

  The process ID returned from MOM is suitable for passing directly to any file-system procedure. (If you expand the process ID into a 12-word array and fill it with blanks on the right before or after the call to MOM, you can pass the process ID as a file name to any Guardian procedure.)

- Calling MOM from an adopted process

  If another process has made itself the creator of the caller of MOM (through a call to STEPMOM or PROCESS_SETINFO_), then the process ID of the adopting process is returned.

- Network consideration

  If a process's creator is on a remote system, its process ID is returned by MOM in network form. A process can use this fact to determine whether it is created locally.

- Calling MOM from a high-PIN process

  If the mom of the calling process is a high-PIN process, MOM returns a synthetic process ID. A synthetic process ID contains a PIN value of 255 in place of a high-PIN value, which cannot be represented by 8 bits.

- Calling MOM from a remote process with a long process name

  If the mom of the calling process is a named process on a remote node and has a process name consisting of more than five characters, the call to MOM fails: a TNS Guardian process terminates with a limits exceeded trap (trap 5); an OSS or native process receives a SIGLIMIT signal.

OSS Considerations

By default, an OSS process does not have a mom process; therefore, zeros are returned in *process-id*. An OSS process can have a mom process if it was created by one of the OSS `tdm_spawn` set of functions, the `tdm_fork()` function, or one of the `tdm_exec` set of functions; see the reference pages either online or in the *Open System Services System Calls Reference Manual* for details. An OSS process does have a mom process if a mom process has been explicitly assigned by either the PROCESS_SETINFO_ or STEPMOM procedure.

MONITORCPUS Procedure

Summary

The MONITORCPUS procedure instructs the operating system to notify the application process if a designated processor module either:

- Fails (indicated by the absence of an operating system "I'm alive" message)
- Returns from a failed to an operable state (that is, reloaded by means of a command interpreter RELOAD command)

The calling application process is notified by means of a system message read through the $RECEIVE file.

Syntax for C Programmers

```
#include <cextdecs(MONITORCPUS)>

void MONITORCPUS ( short cpu-mask );
```

Syntax for TAL Programmers

```
CALL MONITORCPUS ( cpu-mask );          ! i
```

Parameters

cpu-mask input

INT:value

is a bit that is set to "1," corresponding to each processor module to be monitored:

```
<0>   1   processor module 0 to be monitored
<1>   1   processor module 1 to be monitored
.
.
.
<14>  1   processor module 14 to be monitored
<15>  1   processor module 15 to be monitored
      0   means no notification occurs.
```

Messages

- processor down

  System message -2 (processor down) is received if failure occurs with a processor module that is being monitored (for the description and form of system messages, see the *Guardian Procedure Errors and Messages Manual*). Please be aware that this message expires in 3 minutes; it must be read before expiration or it will be lost.

- processor up

  System message -3 (processor up) is received if a reload occurs with a processor module that is being monitored.

For a list of system messages sent to processes, see the *Guardian Procedure Errors and Messages Manual*.

Example

```
CALL MONITORCPUS ( %100000 '>>' BACKUP^CPU );   ! monitor the
                                                ! backup CPU.
```

MONITORNET Procedure

Summary

The MONITORNET procedure enables or disables receipt of system messages concerning the status of processors in remote systems.

Syntax for C Programmers

```
#include <cextdecs(MONITORNET)>

void MONITORNET ( short enable );
```

Syntax for TAL Programmers

```
CALL MONITORNET ( enable );              ! i
```

Parameters

enable input

INT:value

contains one of these values:

0 Disable receipt of messages
1 Enable receipt of messages

Considerations

- To receive status changes for local processors

  MONITORNET only provides notification of status changes for remote processors. To receive notification of status changes for local processors, an application process must still call MONITORCPUS.

- Change in status of network processors

  A process that has enabled MONITORNET receives a system message (-8, -100, -110, -111, or -113) on $RECEIVE whenever a change in the status of a remote processor occurs. The processor status bit masks have a 1 in bit *cpu number* to indicate that the processor is up and a 0 to indicate that the processor is down.

See the *Guardian Procedure Errors and Messages Manual* for details on system messages sent to processes.

MONITORNEW Procedure

Summary

The MONITORNEW procedure enables or disables receipt of the SETTIME and Power On messages.

Syntax for C Programmers

```
#include <cextdecs(MONITORNEW)>

void MONITORNEW ( short enable );
```

Syntax for TAL Programmers

```
CALL MONITORNEW ( enable );            ! i
```

Parameters

enable input

INT:value

contains one of these values:

0 Disable receipt of messages
1 Enable receipt of messages

Consideration

The SETTIME and Power On messages are not received unless the process makes a call to MONITORNEW with *enable* set to 1. To disable receipt of these messages, the process must make another call, setting *enable* to 0.

MOVEX Procedure

Summary

MOVEX moves data between extended data segments without the need for absolute addressing; it serves both privileged and nonprivileged users.

Syntax for C Programmers

```
#include <cextdecs(MOVEX)>

short MOVEX ( [ short source-seg-id ]
              ,char *source
              ,[ short dest-seg-id ]
              ,char *dest
              ,__int32_t byte-count );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
error := MOVEX ( [ source-seg-id ]          ! i
                 ,source                    ! i
                 ,[ dest-seg-id ]           ! i
                 ,dest                      ! i
                 ,byte-count );             ! i
```

Parameters

error returned value

INT

returns a file-system error code indicating the outcome of the call:

0	Successful call; the specified data was moved.
2	Either *source-seg-id* or *dest-seg-id* specified a nonexistent extended data segment, or the destination data segment has read-only access.
22	One of the parameters specifies an address that is out of bounds.

24 Either *source-seg-id* or *dest-seg-id* specified a privileged segment ID (greater than 2047), but the caller was not privileged.
29 A required parameter is not supplied.

source-seg-id input

INT:value

specifies the segment ID of the extended data segment referenced by *source*. If the relative segment in *source* does not indicate an extended data segment, *source-seg-id* is ignored and may be omitted; otherwise it is required and must indicate an existing extended data segment.

source input

STRING .EXT:ref

specifies the relative extended address of the source of the first byte to be moved.

dest-seg-id input

INT:value

specifies the segment ID of the extended data segment referenced by *dest*. If the relative segment in *dest* does not indicate an extended data segment, *dest-seg-id* is ignored and may be omitted. Otherwise, *dest-seg-id* is required and must indicate an existing extended data segment.

dest input

STRING .EXT:ref

specifies the relative extended address of the first byte in the destination location.

byte-count input

INT (32):value

specifies the number of bytes to be moved from *source* to *dest*.

Considerations

- MOVEX works properly only on completely separate selectable segments. The result from a MOVEX operation is undefined if the source and destination ranges overlap. If these ranges overlap, then the move statements ':=' from TAL and '=:' from pTAL should be used instead.
- Segment moves can be performed more efficiently using programming language statements than using MOVEX to move data between:
  - Areas within the same extended data segment
  - Flat extended data segment

  - The currently in-use selectable segment and flat extended data segments
  - nonextended relative locations
  - extended and nonextended relative locations
- If the caller is privileged, no bounds checking is performed.
- This diagram indicates restrictions on data movement from the *source* to the *dest* based upon the privileged state of the caller:

Relative Segment		Source	Destination
0	current data	ok	ok
1	system data	not allowed	not allowed
2	current code	not allowed	not allowed
3	user code	ok	not allowed
>3	extended data seg ID <= 2047	ok	ok
>3	extended data seg ID > 2047	privileged only	privileged only

- MOVEX does not alter the status of the current in-use segment.

Related Programming Manual

For programming information about the MOVEX procedure, see the *Guardian Programmer's Guide*.

MYGMOM Procedure (Superseded by PROCESS_GETINFOLIST_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The MYGMOM procedure provides a process that is a member of a batch job with the process ID of its job ancestor (GMOM). See the *NetBatch User's Guide* for information on batch processing.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
CALL MYGMOM (process-id);                    ! o
```

Parameters

process-id output

INT:ref:4

is a 4-word array where MYGMOM returns the process ID of the job ancestor in network form.

Considerations

- A process may not always have a job ancestor. In that case, zeros are returned.
- The process ID returned from MYGMOM is suitable for passing directly to any file-system procedure. (If you pad the process ID with blanks before or after the call to MYGMOM, you can pass the process ID as a file name to any Guardian procedure.)
- If the job ancestor of the calling process is a high-PIN process, MYGMOM returns a synthetic process ID. A synthetic process ID contains a PIN value of 255 in place of a high-PIN value, which cannot be represented by 8 bits.
- If the job ancestor of the calling process is a named process on a remote node and has a process name consisting of more than five characters, the call to MYGMOM fails: a Guardian TNS process terminates with a limits exceeded trap (trap 5); an OSS or native process receives a SIGLIMIT signal.

Related Programming Manual

For programming information about batch processing and the MYGMOM procedure, see the *NetBatch User's Guide*.

MYPID Procedure (Superseded by PROCESSHANDLE_GETMINE_ Procedure and PROCESSHANDLE_DECOMPOSE_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The MYPID procedure provides a process with its own processor and PIN number. This one-word quantity has been called the PID of a process, with no connection to the 4-word process ID.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
cpu,pin := MYPID;
```

Parameters

cpu,pin returned value

INT

is the caller's processor (bits <4:7>) and PIN number (bits <8:15>). Note that bits <0:3> are always 0.

Considerations

If the caller of the MYPID procedure is a high-PIN process, the call to MYPID fails: a TNS Guardian process terminates with a limits exceeded trap (trap 5); an OSS or native process receives a SIGLIMIT signal.

MYPROCESSTIME Procedure

Summary

The MYPROCESSTIME procedure returns the process execution time of the calling process. Process time is the processor time in microseconds that the process has consumed; processor time used for system procedures called is also included.

Syntax for C Programmers

```
#include <cextdecs(MYPROCESSTIME)>

long long MYPROCESSTIME ();
```

Syntax for TAL Programmers

```
process-time := MYPROCESSTIME;
```

Parameters

process-time returned value

FIXED

is a value representing the clock of the current process in microseconds.

Related Programming Manual

For programming information about the MYPROCESSTIME procedure, see the *Guardian Programmer's Guide*.

MYSYSTEMNUMBER Procedure (Superseded by NODENAME_TO_NODENUMBER_ Procedure or PROCESSHANDLE_GETMINE_ Procedure and PROCESSHANDLE_DECOMPOSE_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The MYSYSTEMNUMBER procedure provides a process with its own system number.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
sysnum := MYSYSTEMNUMBER;
```

Parameters

sysnum returned value

INT

is the caller's system number.

Consideration

- Part of network or local system

  This IF (skeleton) statement determines if you are running on a network system.

```
IF NOT ( SYS^NUM := MYSYSTEMNUMBER ) THEN
   ! not on network system
```

  If the caller is running in a local nonnamed system, MYSYSTEMNUMBER returns 0. Since 0 is a valid system number, a process wishing to determine the name of the system on which it is running can use this call.

```
CALL GETSYSTEMNAME( MYSYSTEMNUMBER, NAME );
```

  A return of all blanks in a name indicates that the system is not part of a network, or that it is a local system which is not named.

MYTERM Procedure (Superseded by PROCESS_GETINFOLIST_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The MYTERM procedure provides a process with the file name of its home terminal. The file name returned from MYTERM is suitable for passing directly to any Guardian procedure that accepts a file name in internal form.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
CALL MYTERM ( file-name );                                    ! o
```

file-name output

INT:ref:12

is a 12-word array where MYTERM returns the device name and the subdevice name, if any, of the home terminal in one of these two forms:

```
$devname [#subdev-name]
$process-name [#subname]
```

Considerations

- The file name returned from MYTERM is the same form as that used by the file-system procedures.
- The home terminal is always the same as the home terminal of a process's true creator (not the process that adopted it through STEPMOM or PROCESS_SETINFO_), unless the home terminal is altered by SETMYTERM or the home terminal option in PROCESS_CREATE_, PROCESS_SPAWN_, NEWPROCESS, NEWPROCESSNOWAIT, OSS `tdm_fork()`, OSS `tdm_spawn()`, or one of the OSS `tdm_exec` set of functions.

  If the process calling MYTERM is a descendant of a command interpreter, then the home terminal is the same as that of the command interpreter or that of an explicit TERM specifier on the RUN command.
- If the home terminal is on a remote node and has either a device name consisting of more than seven characters or a process name consisting of more than five characters, the call to MYTERM fails; a Guardian TNS process terminates with a limits exceeded trap (trap 5): an OSS or native process receives a SIGLIMIT signal. If the home terminal is unnamed and its I/O process is running at a high PIN, MYTERM also fails with a trap 5 or a SIGLIMIT signal.

10 Guardian Procedure Calls (N)

This section contains detailed reference information for all user-accessible Guardian procedure calls beginning with the letter N. Table 10-1 lists all the procedures in this section.

Table 10-1. Procedures Beginning With the Letter N

NEWPROCESS Procedure (Superseded by PROCESS_LAUNCH_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The NEWPROCESS procedure is used to create a new process and, optionally, set a number of process attributes. When a new process is created, its 4-word process ID is returned to the caller.

You can use this procedure to create only Guardian processes, although you can call it from a Guardian process or an OSS process. The program file must contain a program for execution in the Guardian environment. The program file and any user library file must reside in the Guardian name space.

DEFINEs for the process context of the creator can be propagated to a new process. Further, any or all of the file names given in the *filenames* parameter can be DEFINE names.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
CALL NEWPROCESS ( filenames                                  ! i
                 ,[ priority ]                               ! i
                 ,[ memory-pages ]                           ! i
                 ,[ processor ]                              ! i
                 ,[ process-id ]                             ! o
                 ,[ error ]                                  ! o
                 ,[ name ]                                   ! i
                 ,[ hometerm ]                               ! i
                 ,[ flags ]                                  ! i
                 ,[ jobid ]                                  ! i
                 ,[ errinfo ]                                ! o
                 ,[ pfs-size ] );                             ! i
```

Parameters

filenames input

INT:ref:12 or INT:ref:36

is an array that contains the internal-format file name of the program to be run and, optionally, two additional fields. The new process is created on the system where the program file resides. If the program file name is in local form, the caller's system is assumed.

The program file must be in the Guardian name space and contain a program for execution in the Guardian environment.

For the program file only, if you specify a file on the subvolume $SYSTEM.SYSTEM and the file is not found, NEWPROCESS then searches on the subvolume $SYSTEM.SYS*nn*. For information about file names, see Appendix D, File Names and Process Identifiers.

The additional fields, which are used only if bit 1 of the *priority* parameter is set to 1, are as follows:

filenames[12:23] = *library-file*

is the internal-format file name of a user library to be used by the process. The user library must be on the same system as the process being created. If the supplied name is in local form, the system where the process is created is assumed. The library file must reside in the Guardian name space.

filenames[24:35] = *swap-file*

is not used, but you can provide it for informational purposes. If supplied, the swap file must be on the same system as the process being created. If the supplied name is in local form, the system where the process is created is assumed. Native processes swap to a file that is managed by the Kernel-Managed Swap Facility. For more information on this facility, see the *Kernel-*

Managed Swap Facility (KMSF) Manual. To reserve swap space for the process, create the process using the PROCESS_LAUNCH_ procedure and specify the Z^SPACE^GUARANTEE field of the *param-list* parameter. Alternatively, use the nld utility to set native process attributes.

For TNS processes on RVUs preceding the D42 RVU, this field is the internal-format file name of a file to be used as a swap file for the data stack. The swap file must be on the same system as the process being created. If the supplied name is in local form, the system where the process is created is assumed.

For more information, see General Considerations on page 10-18.

priority input

INT:value

is a value consisting of three parts:

<0>	is the debug bit. If *priority*.<0> = 1, the system sets a code breakpoint on the first executable instruction of the program's MAIN procedure.
<1>	determines use of the additional fields of the *filenames* parameter. If *priority*.<1> = 1, the additional fields in *filenames* are used. If *priority*.<1> = 0, these extra fields are ignored.
<2:7>	should be 0.
<8:15>	is the execution priority to be assigned to the new process {1:199}. If *priority*.<8:15> = 0, the priority of the caller of NEWPROCESS is used. If a value greater than 199 is specified, 199 is used.

If *priority* is omitted, the caller's priority is used; this is equivalent to setting bits <0> and <1> to 0.

memory-pages input

INT:value

for TNS processes, specifies the minimum number of 2048-byte memory pages allocated to the new process for user data. The actual amount of memory allocated is processor-dependent. If *memory-pages* is omitted or is less than the value assigned when the program is compiled (or created with Binder), then the compilation value is used. In any case, the maximum number of pages permitted is 64.

For native processes, this parameter is ignored. To specify the maximum size of the main stack, create a new process using the PROCESS_LAUNCH_ procedure and specify the Z^MAINSTACK^MAX field of the *param-list* parameter. Alternatively, use the nld utility to set process attributes.

processor input

INT:value

specifies the processor where the new process runs. If omitted, the new process runs in the same processor as the caller.

process-id output

INT:ref:4

is a four-word array where NEWPROCESS returns the process ID of the new process. If the new process was created in:

- The local system, then the local form of the process ID is returned.
- The remote system, then the network form of the process ID is returned. (A new process is created on the same node where its program file resides.)

If no process was created, zero is returned in *process-id*.

Use a 12-word array if the process ID is to be passed to any of the appropriate file-system procedures that accept file names, such as OPEN, provided if the larger array is blank-filled on the right.

error output

INT:ref:1

returns two numbers indicating the outcome of the process creation attempt. The numbers each occupy one byte in a 16-bit word as follows:

error.<0:7>	*error*
error.<8:15>	*error-detail* (provides additional information about the error)

If the *error* value exceeds 255 (will not fit in 8 bits), it is reported as 119. If the *detail* value exceeds 255, both 8-bit fields contain 119. Because of the limited capacity of this parameter, it has been superseded by the *errinfo* parameter, which returns the full 16-bit value of each number.

Table 10-2 on page 10-7 summarizes the *error* values and relates them to process creation errors (as issued by PROCESS_LAUNCH_) described in Table 12-4 on page 12-120.

name input

INT:ref:3

if present, is a name to be given to the new process. It is entered into the destination control table (DCT). *name* is of the form:

```
name[0:2] = $process-name
```

process-name must be preceded by a dollar sign (“$”) and consists of a maximum of five alphanumeric characters; the first character must be alphabetic. (If the process is created on a remote system and it is necessary to be able to access the process, its name should consist of, at most, four characters and the “$”; this leaves a byte for the system to insert the node number.) Note that $*process-name* is the first 3 words of the 4-word process ID.

If *name* is not supplied, the process ID of the new process is of the unnamed form, containing a timestamp in words [0:2] instead of $*process-name*, with the *cpu,pin* of the new process in the fourth word. The *process-name* will not be entered into the DCT.

hometerm input

INT:ref:12

is the internal-format file name of the home terminal for the new process. The specified value must designate a terminal or a process. The default is the home terminal of the caller.

flags input

INT:value

flags.<10:12> are used to supply the DEFINE mode for the new process:

flags.<10>	0	Use the DEFINE mode of caller
	1	Use value in *flags*.<12>
flags.<12>	0	DEFINEs disabled
	1	DEFINEs enabled

flags.<14:15> set the debugging attributes for the new process:

flags.<14>	1	Saveabend file creation
	0	No saveabend file creation
flags.<15>	1	INSPECT
	0	DEBUG

When *flags* is specified, bits <14> and <15> are ORed with the corresponding flags in the object code file. If *flags*.<14> is set but *flags*.<15> is not, then *flags*.<15> is also set.

If *flags* is omitted, then the defaults are set from the flags in the object code file (set by compiler directives at compile time, after the object flags are ORed with the caller’s debugging attributes).

jobid input

INT:value

is an integer identifying a new job to be created with the new process as the first process of the job and the caller as the GMOM of the new process. This integer is used by the NetBatch Scheduler. (See Batch Processing Considerations on page 10-21)

errinfo output

INT .EXT:ref:2

returns two numbers indicating the outcome of the process creation attempt, as follows:

errinfo[0] *error*

errinfo[1] *error-detail* (provides additional information about the error)

Table 10-2 summarizes the *error* values and relates them to process creation errors (as issued by PROCESS_LAUNCH_) described in Table 12-4 on page 12-120.

pfs-size input

INT(32):value

meaningful only if the process is being created on a pre-G06 RVU. On G06 and later RVUs, this value is range checked, but is otherwise ignored.

If present and nonzero, this parameter specifies the size in bytes of the process file segment (PFS) of the new process. In G-series RVUs, maximum PFS size is 8 MB. In H-series RVUs, maximum PFS size is 32 MB. A value in this range overrides the `nld` or Binder value stored in the program file. If you omit *pfs-size* or specify 0:

- the `nld` or Binder value is used if it is nonzero
- a default value is used otherwise

Table 10-2. Summary of NEWPROCESS Error Codes (page 1 of 12)

error	**Corresponding Process Creation Error (see Table 12-4 on page 12-120)**
<0 : 7>	**<8:15>**
0	No error, process created
1	Process had undefined externals, but was started
2	No process control block available

* For a list of all file-system and DEFINE errors, see the Guardian Procedure Errors and Messages Manual.

**When error.<8:15> indicates the number of a shared run-time library (SRL), the number represents either the public SRL relative number, or 00 for a native user library. For more information on shared run-time libraries (SRLs) see the nld and noft Manual

Table 10-2. Summary of NEWPROCESS Error Codes (page 2 of 12)

error <0 : 7>	Corresponding Process Creation Error (see Table 12-4 on page 12-120) <8:15>	
3	File-system error occurred on program file: <8:15> is file-system error number*	
4	Unable to allocate map	
5	File-system error occurred on swap file: <8:15> is a file-system error number*	
6	Invalid file format:	
	2	Program file is not a disk file.
	3	Library file is not a disk file.
	4	Program file does not have file code 100 or 700.
	5	Library file does not have file code 100 or 700.
	6	Program file does not have correct file structure.
	7	Library file does not have correct file structure.
	8	Program file requires a later RVU of the operating system.
	9	Library file requires a later RVU of the operating system.
	10	Program file does not have a main procedure.
	13	Library file has a main procedure.
	14	Program file has a stack definition of zero pages.
	16	Program file has an invalid procedure entry point (PEP).
	17	Library file has an invalid procedure entry point (PEP).
	18	Initial extended segment information in program file is inconsistent.
	19	Initial extended segment information in library file is inconsistent.
	20	Program file resident size is greater than the code area length.
	21	Library file resident size is greater than the code area length.

* For a list of all file-system and DEFINE errors, see the Guardian Procedure Errors and Messages Manual.

**When error.<8:15> indicates the number of a shared run-time library (SRL), the number represents either the public SRL relative number, or 00 for a native user library. For more information on shared run-time libraries (SRLs) see the nld and noft Manual

Table 10-2. Summary of NEWPROCESS Error Codes (page 3 of 12)

error <0 : 7>	Corresponding Process Creation Error (see Table 12-4 on page 12-120) <8:15>	
6, *continued*	22	The file was not prepared by the nld utility or the Binder program.
	23	Library file was not prepared by the nld utility or the Binder program.
	24	Program file has undefined data blocks.
	25	Library file has undefined data blocks.
	26	Program file has data blocks with unresolved references.
	27	Library file has data blocks with unresolved references.
	28	Program file has too many TNS code segments.
	29	Library file has too many TNS code segments.
	30	Native code length in the program file is invalid.
	31	Native code length in the library file is invalid.
	32	Native code address in the program file is invalid.
	33	Native code address in the library file is invalid.
	34	Native data length in the program file is invalid.
	35	Native data length in the library file is invalid.
	36	Native data address in the program file is invalid.
	37	Native data address in the library file is invalid.
	38	Program file has too many native code segments.
	39	Library file has too many native code segments.
	40	Program file has invalid native resident areas.
	41	Library file has invalid native resident areas.
	42	Accelerator header in program file is invalid.
	43	Accelerator header in library file is invalid.
	44	UC (user code) option was not used when the program file was accelerated.
	45	UL (user library) option was not used when the library file was accelerated.

* For a list of all file-system and DEFINE errors, see the Guardian Procedure Errors and Messages Manual.

**When error.<8:15> indicates the number of a shared run-time library (SRL), the number represents either the public SRL relative number, or 00 for a native user library. For more information on shared run-time libraries (SRLs) see the nld and noft Manual

Table 10-2. Summary of NEWPROCESS Error Codes (page 4 of 12)

error <0 : 7>	Corresponding Process Creation Error (see Table 12-4 on page 12-120) <8:15>	
6, *continued*	46	Program file has entry in native fixup list with invalid external entry point (XEP) index value or invalid code address value.
	47	Library file has entry in native fixup list with invalid external entry point (XEP) index value or invalid code address value.
	48	Accelerated program file has external procedure identifier list (EPIL), internal procedure identifier list (IPIL), or external entry point table with incorrect format.
	49	Accelerated library file has external procedure identifier list (EPIL), internal procedure identifier list (IPIL), or external entry point table with incorrect format.
	50	UC (user code) was accelerated using the wrong Accelerator option (UC, UL, SC, or SL).
	51	UL (user library) was accelerated using the wrong Accelerator option (UC, UL, SC, or SL).
	52	Program file was accelerated with incompatible version of the Accelerator.
	53	Library file was accelerated with incompatible version of the Accelerator.
	54	Program file has invalid callable gateway (GW) table.
	55	Library file has invalid callable gateway (GW) table.
	56	Wrong processor type is target in program file.
	57	Wrong processor type is target in library file.
	58	Program file has inconsistent native fixup list information.
	59	Library file has inconsistent native fixup list information.
	60	An internal structure of the program file contains an error.
6, *continued*	61	An internal structure of the library file contains an error.

* For a list of all file-system and DEFINE errors, see the Guardian Procedure Errors and Messages Manual.

**When error.<8:15> indicates the number of a shared run-time library (SRL), the number represents either the public SRL relative number, or 00 for a native user library. For more information on shared run-time libraries (SRLs) see the nld and noft Manual

Table 10-2. Summary of NEWPROCESS Error Codes (page 5 of 12)

error	Corresponding Process Creation Error (see Table 12-4 on page 12-120)	
<0 : 7>	**<8:15>**	
	62	An internal structure of the program file contains an error.
	63	An internal structure of the library file contains an error.
	64	An internal structure of the program file has an entry point value of 0.
	65	An internal structure of the library file has an entry point value of 0.
	66	An internal structure of the program file contains an error.
	67	An internal structure of the library file contains an error.
	68	The list of unresolved procedure names in the program file contains an error.
	69	The list of unresolved procedure names in the library file contains an error.
	70	The fixup computed an invalid file offset to the code area of the program file.
	71	The fixup computed an invalid file offset to the code area of the library file.
	72	The program file has an invalid fixup item.
	73	The library file has an invalid fixup item.
	74	An internal structure of the program file contains an error.
	75	An internal structure of the library file contains an error.
	76	The program file has an instruction at a call site that is not the type expected for its fixup item.
	77	The library file has an instruction at a call site that is not the type expected for its fixup item.
6, *continued*	78	The header of a native program file is not in correct format.
	79	The header of a native library file is not in correct format.
	80	The code in the program file starts at the wrong virtual address.

* For a list of all file-system and DEFINE errors, see the Guardian Procedure Errors and Messages Manual.

**When error.<8:15> indicates the number of a shared run-time library (SRL), the number represents either the public SRL relative number, or 00 for a native user library. For more information on shared run-time libraries (SRLs) see the nld and noft Manual

Table 10-2. Summary of NEWPROCESS Error Codes (page 6 of 12)

error	Corresponding Process Creation Error (see Table 12-4 on page 12-120)	
<0 : 7>	**<8:15>**	
	81	The code in the library file starts at the wrong virtual address.
	82	The program file has too much data for the main stack.
	84	The code area of the program file is too large.
	85	The code area of the library file is too large.
	86	The program file has a gateway (GW) table but no callable procedures.
	87	The library file has a gateway (GW) table but no callable procedures.
	89	The file codes of the program file and library file do not match.
	90	The program file being started can run only in the Guardian environment and it is being started in the OSS environment, or vice versa.
	91	The library file being started can run only in the Guardian environment and it is being started in the OSS environment, or vice versa.
	92	The program and the library conflict on global data mapping. This error is reported on the program. The user library selected is not compatible with the user library specified when the nld utility or the Binder program originally created the program object file.
	94	The program expects to import variable names from the library and the library is not exporting any.
6, *continued*	96	The program file uses a shared run-time library (SRL) and is switching to a new library, but the program was accelerated by an old version of the Accelerator program that does not support SRL data relocation at fixup time. Use a version of the Accelerator program provided with the D30.00 or later RVU of the operating system.

* For a list of all file-system and DEFINE errors, see the Guardian Procedure Errors and Messages Manual.

**When error.<8:15> indicates the number of a shared run-time library (SRL), the number represents either the public SRL relative number, or 00 for a native user library. For more information on shared run-time libraries (SRLs) see the nld and noft Manual

Table 10-2. Summary of NEWPROCESS Error Codes (page 7 of 12)

error <0 : 7>	Corresponding Process Creation Error (see Table 12-4 on page 12-120) <8:15>	
	97	The library file uses a shared run-time library (SRL) and is switching to a new library, but the program was accelerated by an old version of the Accelerator program that does not support SRL data relocation at fixup time. Use a version of the Accelerator program provided with the D30.00 or later RVU of the operating system.
	98	The program file has no code spaces.
	99	The library file has no code spaces.
	100	The program file is not executable. Either it was not linked with the nld utility or it was not linked correctly.
	101	The library file is not executable. Either it was not linked with the nld utility or it was not linked correctly.
	102	The program file is not executable because it was linked with an incompatible version of the nld utility.
	103	The library file is not executable because it was linked with an incompatible version of the nld utility.
	104	The program file is not executable because it has more than one HP information header. An error occurred during the linking of the program file.
	105	The library file is not executable because it has more than one HP information header. An error occurred during the linking of the library file.
	106	The program file is not executable because it has more than one REGINFO information header. An error occurred during the linking of the program file.
6, *continued*	107	The library file is not executable because it has more than one REGINFO information header. An error occurred during the linking of the library file.
	108	The program file is not executable because it does not have a GINFO information header. An error occurred during the linking of the program file.
	109	The library file is not executable because it does not have GINFO information header. An error occurred during the linking of the library file.

* For a list of all file-system and DEFINE errors, see the Guardian Procedure Errors and Messages Manual.

**When error.<8:15> indicates the number of a shared run-time library (SRL), the number represents either the public SRL relative number, or 00 for a native user library. For more information on shared run-time libraries (SRLs) see the nld and noft Manual

Table 10-2. Summary of NEWPROCESS Error Codes (page 8 of 12)

error <0 : 7>	Corresponding Process Creation Error (see Table 12-4 on page 12-120) <8:15>	
	110	The program file is not executable because it does not have either a HP information header, a REGINFO information header, or a text header. An error occurred during the linking of the program file.
	111	The library file is not executable because it does not have either a HP information header, a REGINFO information header, or a text header. An error occurred during the linking of the library file.
	112	The program file specifies too many shared run-time libraries (SRLs).
	113	The library file specifies too many shared run-time libraries (SRLs).
	114	The program file specifies duplicate shared run-time libraries (SRLs).
	115	The library file specifies duplicate shared run-time libraries (SRLs).
	117	The shared run-time library (SRL) does not export any procedures.
	121	The shared run-time library (SRL) does not have a file code of 700.
7	Unlicensed privileged program.	
8	Process name error:	
	<8:15>	is a file-system error number*.
9	Library conflict.	
10	Unable to communicate with system monitor process:	
	<8:15>	is a file-system error number*.
11	File-system error occurred on library file:	
	<8:15>	is a file-system error number*.
12	Program file and library file specified are same file.	
13	Extended data segment initialization error:	
	<8:15>	is a file-system error number*.
14	Extended segment swap file error:	
	<8:15>	is a file-system error number*.

* For a list of all file-system and DEFINE errors, see the Guardian Procedure Errors and Messages Manual.

**When error.<8:15> indicates the number of a shared run-time library (SRL), the number represents either the public SRL relative number, or 00 for a native user library. For more information on shared run-time libraries (SRLs) see the nld and noft Manual

Table 10-2. Summary of NEWPROCESS Error Codes (page 9 of 12)

error <0 : 7>	Corresponding Process Creation Error (see Table 12-4 on page 12-120) <8:15>
15	Invalid home terminal: <8:15> is a file-system error number*.
16	I/O error to home terminal: <8:15> is a file-system error number*.
17	DEFINE context propagation error: <8:15> is a propagation error number: 0 Unable to convert a DEFINE name to network form (see DEFINE Considerations on page 10-21) 2 Excessive number of DEFINES declared (see DEFINE Considerations on page 10-21) 3 Invalid DEFMODE supplied
18	Object file with an invalid process device subtype (see General Considerations on page 10-18)
19	Process device subtype specified in backup process not the same as that in the primary process
20	DSC error: invalid ZZPIM file (this error is returned only to privileged callers on D-series RVUs)
21	DSC error: dynamic IOP error (this error is returned only to privileged callers on D-series RVUs)
22	*pfs-size* out of range.
23	Cannot create PFS: <8:15> is a file-system error number*.
24	An unknown error number was returned from a remote system (probably running another level of software): <8:15> is an unknown error number*.
25	Unable to allocate a privileged stack for the process
26	Unable to lock the privileged stack for the process
27	Unable to allocate a main stack for the process
28	DSC error: unable to lock the main stack of a native IOP (this error returned only to privileged callers on D-series RVUs)
29	Security inheritance failure

* For a list of all file-system and DEFINE errors, see the Guardian Procedure Errors and Messages Manual.

**When error.<8:15> indicates the number of a shared run-time library (SRL), the number represents either the public SRL relative number, or 00 for a native user library. For more information on shared run-time libraries (SRLs) see the nld and noft Manual

Table 10-2. Summary of NEWPROCESS Error Codes (page 10 of 12)

error <0 : 7>	Corresponding Process Creation Error (see Table 12-4 on page 12-120) <8:15>
30	Unable to allocate the native globals of a native process
31	Unable to lock the native globals of a native IOP (this error returned only to privileged callers)
32	Main stack maximum value too large
33	Heap maximum value too large
34	Space guarantee value too large
35	Process creation request specifies duplicate shared run-time libraries (SRLs); error.<8:15> contains the numbers** of the duplicate SRLs in the form xxyy (where xx is the first SRL and yy is the duplicate SRL)
36	Unable to find a shared run-time library (SRL) specified by the program file; error.<8:15> contains the SRL number** that could not be found
37	Unable to find a shared run-time library (SRL) specified by another SRL; error.<8:15> contains the SRL numbers** in the form xxyy (where xx is the SRL that specifies the yy SRL)
38	Process creation request specifies too many shared run-time libraries (SRLs); error.<8:15> contains the maximum number of SRLs that can be specified
39	The program file requires fixups to a shared run-time library (SRL) that is unavailable because it is running; error.<8:15> contains the SRL number**
40	A shared run-time library (SRL) requires fixups to another SRL that is unavailable because it is running; error.<8:15> contains the SRL numbers** of the two SRLs in the form xxyy (where xx is the SRL that requires the fixup to the running yy SRL)
41	Security violation; Program file is not licensed but a shared run-time library (SRL) is licensed and has instance data; error.<8:15> contains the licensed SRL number**
42	Security violation; Program file is licensed but a shared run-time library (SRL) is not licensed; error.<8:15> contains the unlicensed SRL number**
43	Program file requires a symbol from a shared run-time library (SRL) but the SRL is not exporting it; error.<8:15> contains the SRL number** that does not export the required symbol
47	Requested swap space for the process cannot be guaranteed

* For a list of all file-system and DEFINE errors, see the Guardian Procedure Errors and Messages Manual.

**When error.<8:15> indicates the number of a shared run-time library (SRL), the number represents either the public SRL relative number, or 00 for a native user library. For more information on shared run-time libraries (SRLs) see the nld and noft Manual

Table 10-2. Summary of NEWPROCESS Error Codes (page 11 of 12)

error <0 : 7>	Corresponding Process Creation Error (see Table 12-4 on page 12-120) <8:15>
48	Number of shared run-time libraries (SRLs) specified by a shared run-time library is incorrect; error.<8:15> contains the SRL number** that caused the error
49	A shared run-time library (SRL) has undefined externals; error.<8:15> contains the SRL number** that has undefined externals
50	Number of shared run-time libraries (SRLs) specified for the program file is incorrect
51	Number of shared run-time libraries (SRLs) specified for the library file is incorrect
52	Security violation; a shared run-time library (SRL) must be licensed to be used by callable or privileged code
53	Unable to obtain global virtual space
54	Mismatch between the symbolic reference in the importing module and the actual type in the exporting module
55	There was an unresolved external reference for data
56	Error detail contains these subcodes: 1 IEEE Floating Point unavailable on this CPU 2 Unrecognized floattype in object file 3 Conflicting floattype values in object files
119	Error returned in error.<0:7> too large to fit into one byte; instead of the error parameter, specify the errinfo parameter, which is a two-word parameter, to obtain complete error information
3xx	Invalid file format on shared run-time library (SRL) number** xx.; errinfo contains one of these error subcodes: 1 The file indicated by the DEFINE is not a disk file. 3 The file indicated by the DEFINE does not have the correct file structure. 11 The shared run-time library (SRL) was not prepared by nld utility. 40 The shared run-time library (SRL) code starts at the wrong virtual address, has invalid text, or invalid data. 42 The shared run-time library (SRL) code area is too large.

* For a list of all file-system and DEFINE errors, see the Guardian Procedure Errors and Messages Manual.

**When error.<8:15> indicates the number of a shared run-time library (SRL), the number represents either the public SRL relative number, or 00 for a native user library. For more information on shared run-time libraries (SRLs) see the nld and noft Manual

Table 10-2. Summary of NEWPROCESS Error Codes (page 12 of 12)

error <0 : 7>	Corresponding Process Creation Error (see Table 12-4 on page 12-120) <8:15>	
	43	The shared run-time library (SRL) either has a gateway (GW) table but no callable procedures or has gateways that are not in the (GW) area.
	50	The shared run-time library (SRL) is not executable. Either it was not linked with the nld utility or it was not linked correctly.
	55	The shared run-time library (SRL) is not executable because it does not have either a HP information header, a REGINFO information header, or a text header. An error occurred during the linking of the program file.
	56	The process creation request specifies too many shared run-time libraries (SRLs).
	58	The shared run-time library (SRL) does not export any procedures.
	60	The shared run-time library (SRL) does not have a file code of 700.
5xx	File-system error on shared run-time library (SRL) number xx.**; errinfo contains a file-system error number*	

* For a list of all file-system and DEFINE errors, see the Guardian Procedure Errors and Messages Manual.

**When error.<8:15> indicates the number of a shared run-time library (SRL), the number represents either the public SRL relative number, or 00 for a native user library. For more information on shared run-time libraries (SRLs) see the nld and noft Manual

General Considerations

- When bit 1 of *priority* is set to 1

  To specify only one of the two extra fields, the calling process must set *priority*.<1> to 1 and fill the *file-name* not specified with blanks.

  If *library-file*:

  - is specified, unresolved external references are resolved first from the specified *library-file*, then from the system library.
  - is specified and *library-file*[0] is 0 (binary), then the library file used by the process when it was last run is removed, and the process runs with no library file. (The references that were previously resolved on the user library are resolved on the system library.)
  - is not specified, the program runs with the library file previously associated with the program file, if any. For TNS and non-PIC native programs, but not for PIC

programs, that association can be changed by running the program with a specified library (or by specifying none). The association can be set by the Binder or linker. For more information about TNS user libraries, see the *Binder Manual*. For more information about TNS/R native user libraries and shared run-time libraries, see the *nld and noft Manual*. For more information about TNS/E native user libraries and shared run-time libraries, see the *eld Manual* and the *enoft Manual*. For more information about dynamic-link libraries (including native user libraries used with PIC programs), see the *ld and rld Reference Manual*.

For TNS processes on RVUs preceding the D42 RVU, if *swap-file*:

- is specified and a file of that name exists, that file is used for memory swapping of the user data stack during execution of the process; if no file of that name exists, a file of that name and of the necessary size is created and used for swapping.
- is not specified, a =_DEFAULTS DEFINE *swap-file* is used if available, otherwise a temporary file is created on the disk where the program file resides.
- specifies only the disk device name (filling the rest of the file name with blanks), a temporary file is created on the specified disk device.

- Creation of the backup of a named process pair

  If the backup of a named process pair is created, the backup process becomes the “creator” of the primary (that is, the caller to NEWPROCESS).

- Program file and user library file differences

  A “user library” is an object file containing one or more procedures. The difference between a program file and a library file is that the library file cannot contain a main procedure. Undefined externals from a library are resolved only from the system library. A program file must contain a main procedure. For more information about TNS user libraries, see the *Binder Manual*. For more information about TNS/R native user libraries and shared run-time libraries, see the *nld and noft Manual*. For more information about TNS/E native user libraries and shared run-time libraries, see the *eld Manual* and the *enoft Manual*. For more information about dynamic-link libraries (including native user libraries used with PIC programs), see the *ld and rld Reference Manual*.

- Library conflict—NEWPROCESS error

  The library file for a process can be shared by any number of processes. However, when a TNS or non-PIC native program file is shared by two or more processes, all processes must have the same user library configuration; that is, all processes sharing the program either have the same user library, or they have no user library. An error 9 (“library conflict”) occurs when a copy of the running program runs with a different library configuration than was specified in the call to NEWPROCESS.

- Startup messages and NEWPROCESS

  The caller of NEWPROCESS has the responsibility to format and send a startup message to the new process, if one is required. For more information on the startup message, see the *Guardian Procedure Errors and Messages Manual*.

- Device subtypes for named processes

  Process device subtype is an object file attribute that can be set when compiling or linking a program. FILEINFO, DEVICEINFO, and other information procedures return the device type and subtype of a named process. A process with a device subtype other than zero must be named.

  There are 63 device subtypes available (0 is the default subtype):

  48 - 63 are for general use. Any user may create a named process with a process subtype in this range.

  1 - 47 are reserved for definition by HP. Currently, 1 is a CMI process, 2 is a security monitor process, 30 is a device simulation process, and 31 is a spooler collector process. Additionally, for subtypes 1 - 15, if the caller of NEWPROCESS does not have a creator access ID of the Super ID, the object file is not LICENSED, or the object file is not PROGIDed to the Super ID, NEWPROCESS rejects the request with an error.

- HP reserved process names

  The operating system reserved process name space includes these names: $X*name*, $Y*name*, and $Z*name*, where *name* is from 1 through 4 alphanumeric characters. You should not use names of this form in any application. System-generated process names (from PROCESS_LAUNCH_, PROCESS_SPAWN_, PROCESS_CREATE_ , NEWPROCESS[NOWAIT], PROCESSNAME_CREATE_ , CREATEPROCESSNAME and CREATEREMOTENAME procedures) are selected from this set of names. For more information about reserved process names, see Appendix B, Reserved Process Names.

- Creator access ID (CAID) and process access ID (PAID)

  The creator access ID of the new process is always the same as the process access ID of the creator process. The process access ID of the new process is the same as that of the creator process unless the program file has the PROGID attribute set; in that case the process access ID of the new process is the same as the user ID of the program file's owner and the new process is always local.

- NEWPROCESS and low PINs

  Processes created by NEWPROCESS always have low PINs because a high PIN cannot fit into a 4-word process ID.

DEFINE Considerations

- DEFINEs from the process context of the caller are propagated to the new process. DEFINEs are propagated to the new process according to the DEFINE mode of the new process. Buffer space for DEFINEs being propagated to a new process is limited to 2 MB whether the process is local or remote. However, the caller can propagate only as many DEFINEs as the child's PFS can accommodate in the buffer space for the DEFINEs themselves and in the operational buffer space needed to do the propagation. The maximum number of DEFINEs that can be propagated varies depending upon the size of the DEFINEs being passed. For an estimate of the size of each type of DEFINE, see DEFINESAVE Procedure.
- When a process is created, its DEFINE working set is initialized with the default attributes of class MAP.
- Any or all of the three filenames in the *filenames* parameter may be DEFINE names; NEWPROCESS will use the disk volume or file given in the DEFINE. If *program-file* is a DEFINE name but no such DEFINE exists, the appropriate error is returned. If either of the other names, *library-file* or *swap-file*, is a logical name but the DEFINE is missing, the procedure will behave as if the file name was not present in the call. This characteristic of accepting absence of DEFINEs provides the programmer with a convenient mechanism which allows, but does not require, user specification of library or swap file location.
- Each process has an associated count of the changes to its context. This count is incremented each time the procedures DEFINEADD, DEFINEDELETE, and DEFINEDELETEALL are invoked and a consequent change to the process context occurs. In the case of DEFINEDELETE and DEFINEDELETEALL the count is incremented by one even if more than one DEFINE is deleted. The count is also incremented if the DEFINE mode of the process is changed. If a call to CHECKDEFINE causes a DEFINE in the backup to be altered, deleted or added, then the count for the backup process is incremented. This count is 0 for newly-created processes, and new processes do not inherit the count of their creators.

Batch Processing Considerations

Note. The job ancestor facility is intended for use by the NetBatch product. Other applications that use this facility might be incompatible with the NetBatch product.

- When the process being created is part of a batch job, NEWPROCESS sends a job process creation message to the job ancestor of the batch job. (See the discussion of "job ancestor" in the *Guardian Programmer's Guide*.) The message identifies the new process and contains the job ID as originally assigned by the job ancestor.

  This enables the job ancestor to keep track of all the processes belonging to a given job.

  For the format of the job process creation message, see the *Guardian Procedure Errors and Messages Manual*.

- NEWPROCESS can create a new process and establish that process as a member of the caller's batch job. In that case the caller's job ID is propagated to the new process. If the caller is part of a batch job, to start a new process that is part of the caller's batch job, omit the *jobid* parameter.
- NEWPROCESS can create a new process separate from any batch job, even if the caller is a process that belongs to a batch job. In that case the job ID of the new process is 0. To start a new process that is not part of a batch job, specify 0 for *jobid*.
- NEWPROCESS can create a new batch job and establish the new process as a member of the newly created batch job. In that case, the caller becomes the job ancestor of the new job; the job ID supplied by the caller becomes the job ID of the new process. To start a new batch job, specify a nonzero value for *jobid*.

  A job ancestor must not have a process name that is greater than four characters (not counting the dollar sign). When the caller of NEWPROCESS is to become a job ancestor, it must conform to this requirement.
- When *jobid* is not supplied:
  - If the caller is not part of a batch job, neither is the newly created process; its job ID is 0.
  - If the caller is part of a batch job, the newly created process is part of the same job because its job ID is propagated to the new process.
- Once a process belongs to a batch job, it remains part of the job.

Safeguard Considerations

For information on processes protected by Safeguard, see the *Safeguard Reference Manual*.

OSS Considerations

- You cannot create an OSS process using the NEWPROCESS procedure. NEWPROCESS returns error 12 if you try.
- You can call NEWPROCESS from an OSS process to create a Guardian process.
- Every Guardian process has these security-related attributes for accessing OSS objects. These attributes are passed, unchanged, from the caller to the new process, whether the caller is an OSS process or a Guardian process:
  - Real, effective, and saved user ID
  - Real, effective, and saved group ID
  - Group list
  - Login name

- Current working directory (cwd)
- Maximum file size
- Default OSS file security

No other OSS process attribute is inherited by the new process.

- OSS file opens in the calling process are not propagated to the new process.

Example

```
CALL NEWPROCESS ( pfile^name, , , , process^id, error );
```

Related Programming Manual

For programming information on batch processing, see the *NetBatch User's Guide*.

NEWPROCESSNOWAIT Procedure (Superseded by PROCESS_LAUNCH_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The NEWPROCESSNOWAIT procedure is used to create a new process in a nowait manner and, optionally, set a number of process attributes. When a new process is created, its 4-word process ID is returned to the caller by a system message on the caller's $RECEIVE file.

You can use this procedure to create only Guardian processes, although you can call it from a Guardian process or an OSS process. The program file must contain a program for execution in the Guardian environment. The program file and any user library file must reside in the Guardian name space.

DEFINEs for the process context of the creator can be propagated to a new process. Further, any or all of the filenames given in the *filenames* parameter can be DEFINE names.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
CALL NEWPROCESSNOWAIT ( filenames                  ! i
                       ,[ priority ]               ! i
                       ,[ memory-pages ]           ! i
                       ,[ processor ]              ! i
                       ,[ process-id ]             ! unused
                       ,[ outcome ]                ! o
                       ,[ name ]                   ! i
                       ,[ hometerm ]               ! i
                       ,[ flags ]                  ! i
                       ,[ jobid ]                  ! i
                       ,[ errinfo ]                ! o
                       ,[ pfs-size ] );            ! i
```

Parameters

filenames input

INT:ref:12 or INT:ref:38

is an array that contains the internal-format file name of the program to be run and three additional fields: *library-file*, *swap-file*, and *tag*. The new process is created on the system where the program file resides. If the program file name is in local form, the caller's system is assumed.

The program file must be in the Guardian name space and contain a program for execution in the Guardian environment.

For the program file only, if you specify a file on the subvolume $SYSTEM.SYSTEM and the file is not found, NEWPROCESSNOWAIT then searches on the subvolume $SYSTEM.SYS*nn*. For information about file names, see Appendix D, File Names and Process Identifiers.

The additional fields, which are used only if bit 1 of the *priority* parameter is set to 1, are as follows:

filenames[12:23] = *library-file*

is the internal-format file name of a user library to be used by the process. The user library must be on the same system as the process being created. If the

supplied name is in local form, the system where the process is created is assumed. The library file must reside in the Guardian name space.

filenames[24:35] = *swap-file*

is not used, but you can provide it for informational purposes. If supplied, the swap file must be on the same system as the process being created. If the supplied name is in local form, the system where the process is created is assumed. Processes swap to a file that is managed by the Kernel-Managed Swap Facility. For more information on this facility, see the *Kernel-Managed Swap Facility (KMSF) Manual*. To reserve swap space for the process, create the process using the PROCESS_LAUNCH_ procedure and specify the Z^SPACE^GUARANTEE field of the *param-list* parameter. Alternatively, use the nld utility to set TNS/R native process attributes or the eld utilitiy to set TNS/E native process attributes.

For TNS processes on RVUs preceding the D42 RVU, this field is the internal-format file name of a file to be used as a swap file for the data stack. The swap file must be on the same system as the process being created. If the supplied name is in local form, the system where the process is created is assumed.

filenames[36:37] = *tag*

is a 2-word value used to identify the completion message from the call to NEWPROCESSNOWAIT. See Message.

priority input

INT:value

is a value passed out of *priority* that has three parts:

<0> is the debug bit. If *priority*.<0> = 1, then a code breakpoint is set on the first executable instruction of the program's MAIN procedure.

<1> indicates the interpretation of the additional fields of the *filenames* parameter. If *priority*.<1> = 1, the additional fields in *filenames* are used. If *priority*.<1> = 0, these extra fields are ignored.

<2:7> should be 0.

<8:15> is the execution priority assigned to the new process {1:199}. If *priority*.<8:15> = 0, then the priority of the caller of procedure NEWPROCESSNOWAIT is used. If a value greater than 199 is specified, then 199 is used.

If *priority* is omitted, the caller's priority is used.

memory-pages input

INT:value

For TNS processes, specifies the minimum number of 2048-byte memory pages allocated to the new process for user data. The actual amount of memory allocated is processor-dependent. If *memory-pages* is omitted or is less than the value assigned when the program is compiled (or created with Binder), then the compilation value is used. In any case, the maximum number of pages permitted is 64.

For native processes, this parameter is ignored. To override default values, call the PROCESS_LAUNCH_ procedure to create a new process and specify the Z^MAINSTACK^MAX, Z^HEAP^MAX, and Z^SPACE^GUARANTEE fields of the *param-list* parameter. Alternatively, use the `nld` utility to set the process attributes.

processor input

INT:value

is a value specifying the processor where the new process runs. If omitted, the new process runs in the same processor as the caller.

process-id unused

INT:ref:4

outcome output

INT:ref:1

returns two numbers indicating the outcome of the process creation attempt. The numbers each occupy one byte in a 16-bit word as follows:

outcome.<0:7> *error*
outcome.<8:15> *error-detail* (provides additional information about the error)

If the *error* value exceeds 255 (will not fit in 8 bits), it is reported as 119. If the *detail* value exceeds 255, both 8-bit fields contain 119. Because of the limited capacity of this parameter, it has been superseded by the *errinfo* parameter, which returns the full 16-bit value of each number.

Table 10-2 on page 10-7 summarizes the *error* values and relates them to process creation errors (as issued by PROCESS_LAUNCH_) described in Table 12-4 on page 12-120.

name input

INT:ref:3

if present, is a name to be given to the new process. It is entered into the destination control table (DCT). *name* is of the form:

name[0:2] = $*process-name*

process-name must be preceded by a dollar sign (“$”) and consists of a maximum of five alphanumeric characters; the first character must be alphabetic. (If the process is created in a remote system and it is necessary to be able to access the process, its name should consist of, at most, four characters and the “$”; this leaves a byte for the system to insert the node number into the six bytes above.) Note that $*process-name* is the first 3 words of the 4-word process ID.

If *name* is not supplied, the process ID of the new process is of the unnamed form, containing a timestamp in words [0:2] instead of $*process-name*, with the *cpu,pin* of the new process in the fourth word. The *process-name* will not be entered into the DCT.

hometerm input

INT:ref:12

is the internal-format file name of the home terminal for the new process. The specified value must designate a terminal or a process. The default is the home terminal of the caller.

flags input

INT:value

flags.<10:12> are used to supply the DEFINE mode for the new process:

flags.<10>	0	Use the DEFINE mode of caller
	1	Use value in *flags*.<12>
flags.<12>	0	DEFINEs disabled
	1	DEFINEs enabled

flags.<14:15> set the debugging attributes for the new process:

flags.<14>	1	Saveabend file creation
	0	No saveabend file creation
flags.<15>	1	INSPECT
	0	DEBUG

When *flags* is specified, the bits <14> and <15> are ORed with the corresponding flags in the object code file. If *flags*.<14> is set but *flag*.<15> is not, then *flags*.<15> is also set.

If these *flags* are omitted then the defaults are set from the flags in the object code file (set by compiler directives at compile time, after the object flags are ORed with the caller’s debugging attributes).

jobid input

INT:value

is an integer identifying a new job to be created with the new process as the first process of the job and the caller as the GMOM of the new process. (For batch

processing with NetBatch, see the Batch Processing Considerations subsection under NEWPROCESS Procedure (Superseded by PROCESS_LAUNCH_ Procedure).)

errinfo output

INT .EXT:ref:2

returns two numbers indicating the outcome of the process creation attempt, as follows:

errinfo[0] *error*

errinfo[1] *error-detail* (provides additional information about the error)

Table 10-2 on page 10-7 summarizes the *error* values and relates them to process creation errors (as issued by PROCESS_LAUNCH_) described in Table 12-4 on page 12-120.

pfs-size input

INT(32):value

meaningful only if the process is being created on a pre-G06 RVU. On G06 and later RVUs, this value is range checked, but is otherwise ignored.

If present and nonzero, this parameter specifies the size in bytes of the process file segment (PFS) of the new process.In G-series RVUs, maximum PFS size is 8 MB. In H-series RVUs, maximum PFS size is 32 MB. A value in this range overrides the nld or Binder value stored in the program file. If you omit *pfs-size* or specify 0:

- the nld or Binder value is used if it is nonzero
- a default value is used otherwise

Considerations

- When bit 1 of *priority* is set to 1

  The value in the *tag* parameter appears in the message returned upon completion of NEWPROCESSNOWAIT. To specify only one or two of the three extra fields, the calling process must set *priority*.<1> to 1 and fill the fields not to be specified with blanks.

  If *library-file*:

  - is specified, unresolved external references are resolved first from the specified *library-file*, then from the system library.
  - is specified and *library-file*[0] is 0, then the library file used by the process when it was last run is removed, and the process runs with no library file. (The references that were previously resolved on the user library are resolved on the system library.)

- is not specified, the program runs with the same library file as the last time it was run (or no file, if that was how it was run) or with the library file currently executing.

For TNS processes on RVUs preceding the D42 RVU, if *swap-file*:

- is specified and a file of that name exists, that file is used for memory swaps of the user data stack during execution of the process; if no file of that name exists, a file of that name and of the necessary size is created and used for swaps.
- is not specified, a =_DEFAULTS DEFINE *swap-file* is used if available, otherwise a temporary file is created on the disk where the program file resides.
- specifies only the device name (filling the rest of the file name with blanks), a temporary file is created on the specified device.

- When a nonzero value is returned in *error*

  If NEWPROCESSNOWAIT cannot initiate process creation (for instance, if an invalid processor number is specified), no message appears on $RECEIVE. The *error* parameter is returns a nonzero value indicating the error.

- Startup messages and NEWPROCESSNOWAIT

  The caller of NEWPROCESSNOWAIT has the responsibility to format and send a startup message to the new process, if one is required. For more information on the startup message, see the *Guardian Procedure Errors and Messages Manual*.

- NEWPROCESSNOWAIT and low PINs

  Processes created by NEWPROCESSNOWAIT always have low PINs because a high PIN cannot fit into a 4-word process ID.

- See also subsections General Considerations, DEFINE Considerations, and Batch Processing Considerations under NEWPROCESS Procedure (Superseded by PROCESS_LAUNCH_ Procedure).

NEWPROCESSNOWAIT Completion Message

If NEWPROCESSNOWAIT succeeds in initiating process creation or if an error occurs during process creation, the NEWPROCESSNOWAIT completion system message(-12) is sent to $RECEIVE upon completion. The format of the NEWPROCESSNOWAIT completion message is described in the *Guardian Procedure Errors and Messages Manual*.

DEFINE Considerations

See DEFINE Considerations on page 10-21.

Batch Processing Considerations

See Batch Processing Considerations on page 10-21.

OSS Considerations

- You cannot create an OSS process using the NEWPROCESSNOWAIT procedure. NEWPROCESSNOWAIT returns error 12 if you try.
- You can call NEWPROCESSNOWAIT from an OSS process to create a Guardian process.
- Every Guardian process has these security-related attributes for accessing OSS objects. These attributes are passed, unchanged, from the caller to the new process, whether the caller is an OSS process or a Guardian process:
  - Real, effective, and saved user ID
  - Real, effective, and saved group ID
  - Group list
  - Login name
  - Current working directory (cwd)
  - Maximum file size
  - Default OSS file security

  No other OSS process attribute is inherited by the new process.
- OSS file opens in the calling process are not propagated to the new process.

Example

```
CALL NEWPROCESSNOWAIT ( pfile^name
                        ,              ! Priority.
                        ,              ! Memory pages.
                        ,              ! Processor.
                        ,              ! Process ID - not used
                        ,error
                        ,new^name );
```

Related Programming Manual

For programming information on batch processing, see the *NetBatch User's Guide*.

NEXTFILENAME Procedure (Superseded by FILENAME_FINDNEXT_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The NEXTFILENAME procedure is used to obtain the name of the next disk file on a designated volume. NEXTFILENAME returns the next file name in alphabetic sequence after the file name supplied as the parameter. The alphabetic sequence includes digits 0-9; if the volume contains temporary files, the first temporary file is returned when *file-name* is $*volname* (blank-fill).

The intended use of NEXTFILENAME is in an iterative loop, where the file name returned in one call to NEXTFILENAME specifies the starting point for the alphabetic search in the subsequent call to NEXTFILENAME. In this manner, a volume's file names are returned to the application process in alphabetic order through successive calls to NEXTFILENAME.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
error := NEXTFILENAME ( file-name );      ! i,o
```

Parameters

error returned value

INT

is a file-system error number indicating the outcome of the call. Common errors returned are:

0 No error; next file name in alphabetic sequence is returned in *file-name*.

1 End-of-file, there is no file in alphabetic sequence following the file name supplied in *file-name*.

13 Invalid file name specification.

For a list of all file-system errors, see the *Guardian Procedure Errors and Messages Manual*.

file-name input, output

INT:ref:12

on the call, is the internal-format file name from which the search for the next file name begins. *file-name* on the initial call can be one of these forms.

To obtain the name of the first file on $*volname*:

file-name[0:11]	$*volname* (blank-fill) or *sysnum volname* (blank-fill)

To obtain the name of the first file in *subvol-name* on $*volume*:

file-name[0:3]	$*volname* (blank-fill) or *sysnum volname* (blank-fill)
file-name[4:11] =	*subvol-name* (blank-fill)

To return the name of the next file in alphabetic sequence:

file-name[0:3]	$*volname* (blank-fill) or *sysnum volname* (blank-fill)
file-name[4:7]	*subvol-name* (blank-fill) *file-name*[8:11] *file-id* (blank-fill)

When *file-name* returns, it contains the next file name, if any, in alphabetic sequence.

Considerations

- The NEXTFILENAME procedure can be used to search for files on HP NonStop Storage Management Foundation (SMF) virtual volumes. However, the names in the special SMF subvolumes (ZYS* and ZYT*) where SMF physical files reside are not returned.

Example

```
FNAME ':=' [ "$SYSTEM ", 8 * ["   "]];
WHILE NOT (ERROR := NEXTFILENAME ( FNAME ) ) DO
   BEGIN
      .
      .
      .
   END;
```

NO^ERROR Procedure

Summary

NO^ERROR is called internally by sequential I/O (SIO) procedures. Error handling and retries are implemented within the SIO procedure environment by the NO^ERROR procedure.

If the file is opened by OPEN^FILE, then the NO^ERROR procedure can be called directly for the file-system procedures.

Syntax for C Programmers

```
#include <cextdecs(NO_ERROR)>

short NO_ERROR ( short state
                ,short _near *file-fcb
                ,short _near *good-error-list
                ,short retryable );
```

Syntax for TAL Programmers

```
no-retry := NO^ERROR ( state                    ! i
                      ,file-fcb                 ! i
                      ,good-error-list          ! i
                      ,retryable );             ! i
```

Parameters

no-retry returned value

INT

indicates whether or not the I/O operation should be retried. Values of *no-retry* are:

0	operation should be retried.
<>0	operation should not be retried.

If *no-retry* is not 0, one of this is indicated:

- *state* is not 0.
- No error occurred; error is 0.
- Error is a good error number on the list.
- Fatal error occurred, and abort-on-error mode is OFF.
- Error is a BREAK error, and BREAK is enabled for *file-fcb*.

state input

INT:value

if nonzero, indicates the operation is considered successful. The file error and retry count variables in the file control block (FCB) are set to zero, with *no-retry* returned as nonzero. Typically, either of two values is passed in this position:

= (CCE)	immediately follows a file-system call. If equal is true, the operation is successful. This eliminates a call to FILEINFO by NO^ERROR.
0	forces NO^ERROR to first check the error value in the FCB. If the FCB error is 0, NO^ERROR calls FILEINFO for the file.

file-fcb input

INT:ref:*

identifies the file to be checked.

good-error-list input

INT:ref:*

is a list of error numbers; if one of the numbers matches the current error, *no-retry* is returned as nonzero (no retry). The format of *good-error-list*, in words, is:

[0]	number of error numbers in list {0:n}
[1]	good error number
.	
.	
.	
[n]	good error number

retryable input

INT:value

is used to determine whether certain path errors should be retried. If *retryable* is not zero, errors in the range of {120, 190, 202:231} cause retry according to the device type as follows:

Device	Retry Indication
Operator	Yes
Process	NA
$RECEIVE	NA
Disk	(opened with sync depth of 1, so not applicable)
Terminal	Yes
Printer	Yes
Mag Tape	No

If the path error is either of {200:201}, a retry indication is given in all cases following the first attempt.

Example

```
INT GOOD^ERROR [ 0:1 ] := [ 1, 11 ]; ! nonexistent record.
        .
        .
        .
NO^ERROR ( = , OUT^FILE , GOOD^ERROR , FALSE );
```

NODE_GETCOLDLOADINFO_ Procedure

Summary

The NODE_GETCOLDLOADINFO_ procedure retrieves the name of the OSIMAGE file from which the specified node was system loaded.

NODE_GETCOLDLOADINFO_ assists subsystems that look for their configuration files on $SYSTEM.SYSnn, or that must know the name of the system-load subvolume.

Syntax for C Programmers

```
#include <cextdecs(NODE_GETCOLDLOADINFO_)>

short NODE_GETCOLDLOADINFO_ ( char *filename
                             ,short maxlen
                             ,short *filename-length
                             ,[ const char *nodename ]
                             ,[ short length ] );
```

- The parameter *length* specifies the length in bytes of the character string pointed to by *nodename*. The parameters *nodename* and *length* must either both be supplied or both be absent.

Syntax for TAL Programmers

```
error := NODE_GETCOLDLOADINFO_ ( filename:maxlen          ! o:i
                                ,filename-length          ! o
                                ,[ nodename:length ] ); ! i:i
```

Parameters

error returned value

INT

indicates the outcome of the operation. Valid values are:

0 File name successfully retrieved
1 (reserved)
2 Parameter error
3 Bounds error
4 Unable to communicate with node

filename:*maxlen* output:input

STRING .EXT:ref:*, INT:value

returns the fully qualified name of the file from which the specified node was system loaded.

maxlen is the length in bytes of the string variable *filename*.

filename-length output

INT .EXT:ref:1

is the actual length in bytes of the returned file name.

nodename:*length* input:input

STRING .EXT:ref:*, INT:value

if supplied and *length* is not 0, specifies the name of the node for which system-load information is to be returned. If used, the value of *nodename* must be exactly *length* bytes long. The default is the name of the local node.

Example

```
error := NODE_GETCOLDLOADINFO_ ( name:maxlen, name^len );
```

NODENAME_TO_NODENUMBER_ Procedure

Summary

The NODENAME_TO_NODENUMBER_ procedure converts a node name (system name) to the corresponding node number (system number). It can also be used to obtain the number of the caller's node.

Syntax for C Programmers

Note. In the TNS/E environment the CEXTDECS file uses the *int* data type for 32-bit values. This is a change from the TNS and TNS/R environments where CEXTDECS uses the *long* data type for 32-bit values.

```
#include <cextdecs(NODENAME_TO_NODENUMBER_)>

short NODENAME_TO_NODENUMBER_ ( [ const char *nodename ]
                              ,[ short length ]
                              ,__int32_t *nodenumber
                              ,[ __int32_t *ldevnum ]);
```

The parameter *length* specifies the length in bytes of the character string pointed to by *nodename*. The parameters *nodename* and *length* must either both be supplied or both be absent.

Syntax for TAL Programmers

```
error := NODENAME_TO_NODENUMBER_ ( [ nodename:length ]   ! i:i
                                  ,nodenumber            ! o
                                  ,[ ldevnum ] );        ! o
```

Parameters

error returned value

INT

is a file-system error number indicating the outcome of the operation.

nodename:*length* input:input

STRING .EXT:ref:*, INT:value

specifies the name of the node whose number is to be returned. *nodename* must be exactly *length* bytes long. If *nodename* is omitted or if *length* is 0, the number of the local node is returned.

nodenumber output

INT(32) .EXT:ref:1

returns the number of the specified node. If *nodename* is omitted or if *length* is 0, *nodenumber* returns the number of the caller's node.

ldevnum output

INT(32) .EXT:ref:1

returns the logical device number of the line handler to the specified node. If the specified node is the local node, *ldevnum* returns 32767. If *error* is nonzero, *ldevnum* is undefined.

NODENUMBER_TO_NODENAME_ Procedure

Summary

The NODENUMBER_TO_NODENAME_ procedure converts a node number (system number) to the corresponding node name (system name). It can also be used to obtain the name of the caller's node.

Syntax for C Programmers

```
#include <cextdecs(NODENUMBER_TO_NODENAME_)>

short NODENUMBER_TO_NODENAME_ ( [ __int32_t nodenumber ]
                                ,char *nodename
                                ,short maxlen
                                ,short *length
                                ,[ __int32_t *ldevnum ] );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
error := NODENUMBER_TO_NODENAME_ ( [ nodenumber ]       ! i
                                  ,nodename:maxlen      ! o:i
                                  ,nodename-length      ! o
                                  ,[ ldevnum ] );       ! o
```

Parameters

error returned value

INT

is a file-system error number indicating the outcome of the operation.

nodenumber input

INT(32):value

if present and not -1D, is the number of the node whose name is to be returned. If *nodenumber* is omitted or -1D, the name of the caller's node is returned.

nodename:*maxlen* output:input

STRING .EXT:ref:*, INT:value

returns the name of the specified node. If *nodenumber* is omitted, *nodename* returns the name of the caller's node. *maxlen* specifies the length in bytes of the string variable *nodename*.

nodename-length output

INT .EXT:ref:1

returns the length in bytes of the value returned in *nodename*.

ldevnum output

INT(32) .EXT:ref:1

returns the logical device number of the line handler to the specified node. If the specified node is the local node, *ldevnum* returns 32767. If *error* is nonzero, *ldevnum* is undefined.

Consideration

If the value specified for *nodenumber* does not designate a node that is known to the local system, an *error* value of 18 is returned. In this case, the *nodename* parameter returns a printable string such as "\255" showing the node number that was supplied as input (provided that the number fits in seven characters).

NSK_FLOAT_IEEE TO TNS Procedures

NSK_FLOAT_IEEE32_TO_TNS32_ Procedure
NSK_FLOAT_IEEE64_TO_TNS32_ Procedure
NSK_FLOAT_IEEE64_TO_TNS64_ Procedure

Summary

These procedures convert numbers in the IEEE floating-point format to numbers in the TNS floating-point format.

The specific functions of each are as follows:

Procedure	Function
NSK_FLOAT_IEEE32_TO_TNS32_	Convert a 32-bit IEEE floating-point value to a 32-bit TNS floating-point value.
NSK_FLOAT_IEEE64_TO_TNS32_	Convert a 64-bit IEEE floating-point value to a 32-bit TNS floating-point value.
NSK_FLOAT_IEEE64_TO_TNS64_	Convert a 64-bit IEEE floating-point value to a 64-bit TNS floating-point value.

Note. These procedures are supported in the G06.06 RVU and all subsequent G-series RVUs. IEEE floating-point is available on all S-series processors except S70000 servers with NSR-G processors.

Syntax for C Programmers

```
#include <kfpconv.h>

uint32 NSK_FLOAT_IEEE32_TO_TNS32_
      ( const NSK_float_IEEE32 *in_p  /* pointer to input */
      , NSK_float_TNS32        *out_p /* pointer to output */
      );

uint32 NSK_FLOAT_IEEE64_TO_TNS32_
      ( const NSK_float_IEEE64 *in_p  /* pointer to input */
      , NSK_float_TNS32        *out_p /* pointer to output */
      );

uint32 NSK_FLOAT_IEEE64_TO_TNS64_
      ( const NSK_float_IEEE64 *in_p  /* pointer to input */
      , NSK_float_TNS64        *out_p /* pointer to output */
      );
```

Syntax for TAL Programmers

```
ErrorBits := NSK_FLOAT_IEEE32_TO_TNS32_ ( IEEEData    ! i
                                         ,TNS_Data ); ! o

ErrorBits := NSK_FLOAT_IEEE64_TO_TNS32_ ( IEEEData    ! i
                                         ,TNS_Data ); ! o

ErrorBits := NSK_FLOAT_IEEE64_TO_TNS64_ ( IEEEData    ! i
                                         ,TNS_Data ); ! o
```

Parameters

ErrorBits returned value

INT(32)

returns the 32-bit error mask.

No bits set means the result was exactly equal in value to the input. This value can be identified with NSK_FLOAT_OK, which is equal to zero.

This is a list of the error bits that can be set by at least one of the three IEEE_TO_TNS conversion procedures:

NSK_FLOAT_TNS_OVERFLOW	The input was out of range (either too big in magnitude, infinity, or not a number (NaN)). The result had the largest possible magnitude.
NSK_FLOAT_TNS_UNDERFLOW	The input was out of range (too small in magnitude) and could not be represented correctly.
NSK_FLOAT_TNS_INEXACT	The result did not exactly match the input.
NSK_FLOAT_WAS_INFINITY	Overflow happened because the input was an IEEE infinity.
NSK_FLOAT_WAS_NAN	Overflow happened because the input was an IEEE NaN.

This indicates which procedures can produce which errors:

Conversion	Over-flow	Under-flow	In-exact	Was_ Inf	Was_ NaN
IEEE64 to TNS64	YES	YES	YES	YES	YES
IEEE64 to TNS32	YES	YES	YES	YES	YES
IEEE32 to TNS32	YES	NO	YES	YES	YES

Note. For IEEE32-to-TNS32 conversion, overflow can occur only if the input is infinite or a NaN.

For NSK_FLOAT_IEEE32_TO_TNS32_:

IEEE_Data input

```
INT .ext:ref (NSK_float_ieee32)
```

The 32-bit IEEE floating-point number.

TNS_Data output

```
INT .ext:ref (NSK_float_tns32)
```

The 32-bit TNS floating-point number

For NSK_FLOAT_IEEE32_TO_TNS64_:

IEEE_Data input

INT .ext:ref (NSK_float_ieee32)

The 32-bit IEEE floating-point number.

TNS_Data output

INT .ext:ref (NSK_float_tns64)

The 64-bit TNS floating-point number

For NSK_FLOAT_IEEE64_TO_TNS64_:

IEEE_Data input

INT .ext:ref (NSK_float_ieee64)

The 64-bit IEEE floating-point number.

TNS_Data output

INT .ext:ref (NSK_float_tns64)

The 64-bit TNS floating-point number

Considerations

- These procedures are usable by both TNS floating-point-format callers and IEEE floating-point-format callers.
- The procedures do not require the data to be aligned on 4-byte or 8-byte boundaries. Shared2 (2-byte) alignment is sufficient.
- The NonStop operating system uses big-endian data formats for all data. For data interchange with little-endian computers using IEEE floating point (such as Alpha processors and Intel® processors), you must reverse the order of bytes in the data.
- Four data structures are declared for containers of data in the four supported formats:

NSK_float_ieee64	For 64-bit IEEE floating-point numbers
NSK_float_tns64	For 64-bit TNS floating-point numbers
NSK_float_ieee32	For 32-bit IEEE floating-point numbers
NSK_float_tns32	For 32-bit TNS floating-point numbers

Examples

C Example

```
#include <kfpconv.h>
#include <stdio.h>

void example1(void) {

   NSK_float_ieee64 before;
   NSK_float_tns32 after;

   ReadIEEE64(&before); /* read in value to convert */
   if( NSK_FLOAT_IEEE64_TO_TNS32_( &before, &after )
      & NSK_FLOAT_TNS_OVERFLOW )

      printf( "Overflow!\n" );

   WriteTNS32(&after); /* write out result */

}
```

TAL Example

```
?nolist
?source $system.system.kfpconv
?list

int(32) proc example2( x );
real(64) .ext x; -- IEEE64 before, TNS64 after

begin

   int(32) error;
   int .ext before ( NSK_float_ieee64 ) = x;
   int .ext after ( NSK_float_tns64 ) = x;

   error := NSK_FLOAT_IEEE64_TO_TNS64_( before, after );
   if ($int(error) LAND $int(NSK_FLOAT_TNS_OVERFLOW)) then
      return( 2D ); -- 2 for overflow (out of range)
   return( 0D ); -- 0 for no errors

end;
```

NSK_FLOAT_TNS TO IEEE Procedures

NSK_FLOAT_TNS32_TO_IEEE32_ Procedure
NSK_FLOAT_TNS32_TO_IEEE64_ Procedure
NSK_FLOAT_TNS64_TO_IEEE64_ Procedure

Summary

These procedures convert numbers in the TNS floating-point format to numbers in the IEEE floating-point format.

The specific functions of each are as follows:

Procedure	Function
NSK_FLOAT_TNS32_TO_IEEE32_	Convert a 32-bit TNS floating-point value to a 32-bit IEEE floating-point value.
NSK_FLOAT_TNS32_TO_IEEE64_	Convert a 32-bit TNS floating-point value to a 64-bit IEEE floating-point value.
NSK_FLOAT_TNS64_TO_IEEE64_	Convert a 64-bit TNS floating-point value to a 64-bit IEEE floating-point value.

Note. These procedures are supported in the G06.06 RVU and all subsequent G-series RVUs. IEEE floating-point is available on all S-series processors except S70000 servers with NSR-G processors.

Syntax for C Programmers

```
#include <kfpconv.h>

uint32 NSK_FLOAT_TNS32_TO_IEEE32_
      ( const NSK_float_TNS32  *in_p  /* pointer to input */
      , NSK_float_IEEE32       *out_p /* pointer to output */
      );

uint32 NSK_FLOAT_TNS32_TO_IEEE64_
      ( const NSK_float_TNS32  *in_p  /* pointer to input */
      , NSK_float_IEEE64       *out_p /* pointer to output */
      );

uint32 NSK_FLOAT_TNS64_TO_IEEE64_
      ( const NSK_float_TNS64  *in_p  /* pointer to input */
      , NSK_float_IEEE64       *out_p /* pointer to output */
      );
```

Syntax for TAL Programmers

```
ErrorBits := NSK_FLOAT_TNS32_TO_IEEE32_ ( TNS_Data    ! i
                                         ,IEEE_Data ); ! o

ErrorBits := NSK_FLOAT_TNS32_TO_IEEE64_ ( TNS_Data     ! i
                                         ,IEEE_Data ); ! o

ErrorBits := NSK_FLOAT_TNS64_TO_IEEE64_ ( TNS_Data     ! i
                                         ,IEEE_Data ); ! o
```

Parameters

ErrorBits returned value

INT(32)

returns the 32-bit error mask.

No bits set means the result was exactly equal in value to the input. This value can be identified with NSK_FLOAT_OK, which is equal to zero.

This is a list of the error bits that can be set by at least one of the three TNS_TO_IEEE conversion procedures:

NSK_FLOAT_IEEE_OVERFLOW	The input was out of range (too big in magnitude), and the result was an IEEE infinity. The sign of the result matched the sign of the input.
NSK_FLOAT_IEEE_UNDERFLOW	The input was out of range (too small in magnitude) and could not be represented exactly, even as a denormalized number.
NSK_FLOAT_IEEE_INEXACT	The result did not exactly match the input.

This indicates which procedures can produce which errors:

Conversion	Overflow	Underflow	Inexact
TNS64 to IEEE64	NO	NO	YES
TNS32 to IEEE64	NO	NO	NO
TNS32 to IEEE32	YES	YES	YES

For NSK_FLOAT_TNS32_TO_IEEE32_

TNS_Data input

```
INT .ext:ref NSK_float_tns32
```

The 32-bit TNS floating-point number.

IEEE_Data output

```
INT .ext:ref NSK_float_ieee32
```

The 32-bit IEEE floating-point number

For NSK_FLOAT_TNS32_TO_IEEE64_

TNS_Data input

```
INT .ext:ref NSK_float_tns32
```

The 32-bit TNS floating-point number.

IEEE_Data output

```
INT .ext:ref NSK_float_ieee64
```

The 64-bit IEEE floating-point number

For NSK_FLOAT_TNS64_TO_IEEE64_

TNS_Data input

```
INT .ext:ref NSK_float_tns64
```

The 64-bit TNS floating-point number.

IEEE_Data output

INT .ext:ref NSK_float_ieee64

The 64-bit IEEE floating-point number

Considerations

For description of considerations for this procedure, ee Considerations on page 10-43.

Examples

C Example

```
#include <kfpconv.h>
#include <stdio.h>

void example3(void) {

   NSK_float_tns32 before;
   NSK_float_ieee32 after;

   ReadTNS32(&before); /* read in value to convert */

   if( NSK_FLOAT_TNS32_TO_IEEE32_( &before, &after )
     & NSK_FLOAT_IEEE_OVERFLOW )

      printf( "Overflow!\n");

   WriteIEEE32(&after); /* write out result */

}
```

TAL Example

```
?nolist
?source $system.system.kfpconv
?list

int(32) proc example4( x );
real(64) .ext x; -- TNS64 before, IEEE64 after

begin

   int(32) error;
   int .ext before ( NSK_float_tns64 ) = x;
   int .ext after ( NSK_float_ieee64 ) = x;

   error := NSK_FLOAT_TNS64_TO_IEEE64_( before, after );

   if ($int(error) LAND $int(NSK_FLOAT_IEEE_INEXACT)) then
      return( 1D ); -- 1 for inexact
   return( 0D ); -- 0 for no errors

end;
```

NUMBEREDIT Procedure

Summary

The NUMBEREDIT procedure renumbers the lines of an EDIT file that are in a specified range. You can specify the new starting number and increment for the range of lines to be renumbered.

NUMBEREDIT is an IOEdit procedure and can only be used with files that have been opened by OPENEDIT or OPENEDIT_.

Syntax for C Programmers

```
#include <cextdecs(NUMBEREDIT)>

short NUMBEREDIT ( short filenum
                  ,__int32_t first
                  ,__int32_t last
                  ,[ __int32_t start ]
                  ,[ __int32_t increment ] );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
error := NUMBEREDIT ( filenum                ! i
                     ,first                  ! i
                     ,last                   ! i
                     ,[ start ]              ! i
                     ,[ increment ] );       ! i
```

Parameters

error returned value

INT

returns a value that indicates the outcome of the operation. The value is a file-system error number or one of these values:

-6 Exhausted valid line numbers.
-10 Unable to complete renumbering; file is unchanged.

filenum input

INT:value

is the number that identifies the open file in which lines are to be renumbered.

first input

INT(32):value

specifies 1000 times the line number of the first line in the range of lines to be renumbered. If a negative value is specified, the line number of the first line in the file is used.

last input

INT(32):value

specifies 1000 times the line number of the last line in the range of lines to be renumbered. If a negative value is specified, the line number of the last line in the file is used.

start input

INT(32):value

specifies 1000 times the line number to be assigned to the first renumbered line. If this parameter is omitted, the old line number is retained for the first renumbered line unless *first* is a negative value, in which case 1000 is used for *start*.

increment input

INT(32):value

specifies 1000 times the value to be added to each successive line number when renumbering lines. If this parameter is omitted, 1000 is used unless the value represented by *start* has a fractional part (that is, if *start*, when divided by 1000, contains a value to the right of the decimal point); in that case, the value used is the largest power of 10 that does not exceed the value of the fractional part. If *last* is a negative value, 1000 is used for *increment*.

Example

In this example, NUMBEREDIT renumbers lines 50 through 100 in the specified file. After the call, these same lines will be numbered starting at 49 with successive line numbers increasing by an increment of 0.100.

```
INT(32) first := 50000D;
INT(32) last := 100000D;
INT(32) start := 49000D;
```

```
INT(32) increment := 100D;
      .
      .
err := NUMBEREDIT ( filenumber, first, last,
                    start, increment );
```

Related Programming Manual

For programming information about the NUMBEREDIT procedure, see the *Guardian Programmer's Guide*.

NUMIN Procedure

Summary

The NUMIN procedure converts the ASCII characters used to represent a number into the signed integer value for that number.

Syntax for C Programmers

```
#include <cextdecs(NUMIN)>

short NUMIN ( char *ascii-num
             ,short _near *signed-result
             ,short base
             ,short _near *status );
```

Syntax for TAL Programmers

```
next-addr := NUMIN ( ascii-num                ! i
                    ,signed-result            ! o
                    ,base                     ! i
                    ,status );                ! o
```

Parameters

next-addr returned value

BADDR

returns the ‘G’[0] relative string address of the first character in *ascii-num* not used in the conversion.

ascii-num input

STRING:ref:*

is an array containing the number to be converted to signed integer form. *ascii-num* is of the form:

```
[ + ] [ % ] [ h/H ]      number nonnumeric
[ - ]       [ b/B ]
```

where “%” means treat the number as a binary, octal, or hexadecimal value (as indicated) regardless of the specified *base*. Note that *nonnumeric* applies only to hexadecimal values.

signed-result output

INT:ref:1

returns the result of the conversion.

base input

INT:value

specifies the number base of *ascii-num*. Legitimate values are 2 through 10 and 16.

status output

INT:ref:1

returns a number that indicates the outcome of the conversion. The values for *status* are:

 1 Nonexistent number (string does not start with “+,” “-,” “%,” or numeric)
 0 Valid conversion
-1 Invalid integer (number cannot be represented in 15 bits) or bad character in *ascii-num*.

Considerations

- When number conversion stops

  Number conversion stops on the first ASCII numeric character representing a value greater than *base* -1 or a nonnumeric ASCII character.

- Base-10 numeric value range

  Base-10 numeric values must be in the range of -32768 through 32767. Numeric values in other number bases are accepted if they can be represented in 16 bits. Note that the magnitude is computed first, so the value can then be negated (for example, %177777 = -%1).

Related Programming Manual

For programming information about the NUMIN procedure, see the *Guardian Programmer's Guide*.

NUMOUT Procedure

Summary

The NUMOUT procedure converts unsigned integer values to their ASCII equivalents. The result is returned right-justified in an array. Any preceding blanks are zero filled.

Syntax for C Programmers

```
#include <cextdecs(NUMOUT)>

void NUMOUT ( char *ascii-result
              ,short unsigned-integer
              ,short base
              ,short width );
```

Syntax for TAL Programmers

```
CALL NUMOUT ( ascii-result                  ! o
             ,unsigned-integer              ! i
             ,base                          ! i
             ,width );                      ! i
```

Parameters

ascii-result output

STRING:ref:*

is an array where the converted value returns. The ASCII representation is right-justified in *ascii-result*[0:*width* -1]. Any preceding blanks are zero filled.

unsigned-integer input

INT:value

is the value to be converted.

base input

INT:value

is the number base for the resulting conversion. Any number in the range 2 to 10 is valid.

width input

INT:value

is the maximum number of characters permitted in *ascii-result*. Characters might be truncated on the left side.

Consideration

If *width* is too small to contain the number, the most significant digits are lost.

Related Programming Manual

For programming information about the NUMOUT utility procedure, see the *Guardian Programmer's Guide*.

11 Guardian Procedure Calls (O)

This section contains detailed reference information for all user-accessible Guardian procedure calls beginning with the letter O. Table 11-1 lists all the procedures in this section.

Table 11-1. Procedures Beginning With the Letter O

OBJFILE_GETINFOLIST_ Procedure

Summary

The OBJFILE_GETINFOLIST_ procedure obtains information about the object file or user library file of the calling process.

Note. OBJFILE_GETINFOLIST_ does not support dynamic-link libraries.

Syntax for C Programmers

```
#include <cextdecs(OBJFILE_GETINFOLIST_)>

short OBJFILE_GETINFOLIST_ ( short *ret-attr-list
                            ,short ret-attr-count
                            ,short *ret-values-list
                            ,short ret-values-maxlen
                            ,short *ret-values-len
                            ,[ short lib-info ]
                            ,[ short *error-detail ]
                            ,[ const char  *srl-filename ]
                            ,[ short srl-filename-len ] );
```

Syntax for TAL Programmers

```
error := OBJFILE_GETINFOLIST_
             ( ret-attr-list                          ! i
              ,ret-attr-count                         ! i
              ,ret-values-list                        ! o
              ,ret-values-maxlen                      ! i
              ,ret-values-len                         ! o
              ,[ lib-info ]                           ! i
              ,[ error-detail ]                       ! o
              ,[ srl-filename:srl-filename-len ] );  ! i:i
```

Parameters

error returned value

INT

indicates the outcome of the operation. It returns one of these values:

0 Information is returned successfully.

1 File-system error; *error-detail* contains the error number. Error 563 (buffer too small) is returned if *ret-values-list* is too small to contain all the requested information.

2 Parameter error; *error-detail* contains the number of the first parameter found to be in error, where 1 designates the first parameter on the left.

3 Bounds error; *error-detail* contains the number of the first parameter found to be in error, where 1 designates the first parameter on the left.

4 Invalid attribute code specified; *error-detail* contains the attribute code that is unknown to OBJFILE_GETINFOLIST_.

5 The process does not have a user library.

6 The process header cannot be found.

ret-attr-list input

INT .EXT:ref:*

specifies an array of INTs indicating the attributes that are to have their values returned in *ret-values-list*. For details, see Attribute Codes and Value Representations on page 11-5.

ret-attr-count input

INT:value

specifies how many items the caller is supplying in *ret-attr-list*.

If the requested information doesn't fit in *ret-values-list*, the procedure returns an *error* value of 1 and an *error-detail* value of 563 (buffer too small). No information is returned.

The maximum value for this parameter is 1024.

ret-values-list output

INT .EXT:ref:*

contains *ret-values-len* bytes of returned information. The values parallel the items in *ret-attr-list*. Each value begins on a word boundary. For details, see Attribute Codes and Value Representations on page 11-5.

ret-values-maxlen input

INT:value

specifies the maximum length in bytes of *ret-values-list*. The size of *ret-values-list* cannot exceed 1024 bytes.

ret-values-len output

INT .EXT:ref:1

returns the actual length in bytes of *ret-values-list*.

lib-info input

INT:value

specifies whether you are requesting information on an object file or a library file.

0 (default)	Object file
1	TNS user library file
2	Native shared run-time library file

error-detail output

INT .EXT:ref:1

for some error conditions, contains additional information. See the returned value *error*.

srl-filename:*srl-filename-len* input:input

if *lib-info* is 2, specifies the name of the native shared run-time library. The value of *srl-filename* must be exactly *srl-filename-len* bytes long. This parameter is ignored if *lib-info* is 0 or 1. To obtain the name of a native shared run-time library, call the PROCESS_GETINFOLIST_ procedure with attributes 115 through 118.

General Consideration

If an error is returned, the contents of *ret-values-list* and *ret-values-len* are undefined.

Attribute Codes and Value Representations

The individual attribute codes and their associated value representations are as follows:

Attribute Code		TAL Value Representation
1	Binder timestamp (for TNS object files only)	INT (3 words)
2	minimum tosversion (for TNS object files only)	INT
3	Inspect length (for TNS object files only)	INT(32)
4	Binder length (for TNS object files only)	INT(32)
5	Inspect on	INT
6	high PIN	INT
7	high requesters	INT
8	run named	INT
9	PFS size	INT(32)
10	target processor	INT
11	accelerator timestamp (for TNS object files only)	INT (4 words)
12	compilation mode (for TNS object files only)	INT
13	run mode (for TNS object files only)	INT
15	`linker` timestamp (for native object files only)	INT(32)
16	buffer size for attribute code 17 (for non-PIC native object files only)	INT
17	Native shared run-time library name information (in a variable-sized array) (for non-PIC native object files only)	INT number of names, INT file name length, STRING name

Each value begins on a word boundary. The attribute values are:

- 1: Binder timestamp (for TNS object files only)

  is the three-word timestamp for when the object file was last updated. For a description of this timestamp, see TIMESTAMP Procedure. If the object file is a native object file, 0 is returned.

- 2: minimum tosversion (for TNS object files only)

is the earliest RVU of the operating system on which the object file can run. For a description of the tosversion, see TOSVERSION Procedure. A value of 0 indicates that either the object file can run on any RVU of the operating system or the object file is a native object file.

- 3: Inspect length (for TNS object files only)

  is the length in bytes of the Inspect region of the file. A value of 0 is returned if the file has no Inspect region. If the object file is a native object file, 0D is returned.

- 4: Binder length (for TNS object files only)

  is the length in bytes of the Binder region of the file. A value of 0 is returned if the file has no Binder region. If the object file is a native object file, 0D is returned.

- 5: Inspect on

  indicates whether the debugger for the file is the Inspect debugger or Debug. A value of 0 indicates Debug; a value of 1 indicates the Inspect debugger.

- 6: high PIN

  indicates whether the process can run with a high PIN. A value of 1 indicates it can run with a high PIN; a value of 0 indicates it must be a low PIN. If either the object file or its user library has the high PIN flag turned off, the process must run with a low PIN.

- 7: high requesters

  indicates whether the process can handle requests from high-PIN processes. A value of 1 indicates that the process can handle requests from high-PIN processes; a value of 0 indicates that the process might or might not support requests from high-PIN processes.

- 8: run named

  indicates whether the object file must be run as a named process. A value of 1 indicates that the object file must be run as a named process; a value of 0 indicates that the object file is not required to run as a named process. If either the object file or its user library has this attribute set to 1, the process is given a name even if none is explicitly requested by the creator.

- 9: PFS size

  is the size in words of the process file segment (PFS) as specified in the object file. If value is 0, the `nld` or Binder value is used if it is nonzero; otherwise, a default value is used.

- 10: target processor

  indicates the processor family for which the program has been compiled. Possible values are:

  0 Unspecified
  1 TNS/R or TNS/E processors

2 TNS processors
3 Any

If the object file is a native object file, the value of this attribute is always 1 (TNS/R or TNS/E processors).

- 11: accelerator timestamp (for TNS object files only)

  is the four-word Julian timestamp for when the object file was accelerated. For a description of the Julian timestamp, see JULIANTIMESTAMP Procedure. If the file is not accelerated or the file is a native object file, 0 is returned. The accelerator timestamp should not be confused with the accelerator version timestamp.

- 12: compilation mode (for TNS object files only)

  indicates whether the object file has been accelerated. A value of 1 indicates that it has been accelerated; a value of 0 indicates that it has not been accelerated.

- 13: run mode (for TNS object files only)

  indicates whether the object file will run accelerated. A value of 1 indicates that it will run accelerated; a value of 0 indicates that it will not run accelerated. Run mode is meaningful only if the file has been accelerated.

- 15: `linker` timestamp (for native object files only)

  is the 32-bit integer timestamp in the form returned by the `time()` function defined in the header file `time.h`. That is, it is a UNIX-style timestamp. This form, the Coordinated Universal Time, is expressed as the number of seconds since the start of January 1, 1970. If the file is not a native object file, 0D is returned.

- 16: buffer size for attribute code 17 (for non-PIC native object files only)

  is the size of the buffer, in bytes, for the array returned in attribute 17. 0 indicates that the object file does not use shared run-time libraries.

- 17: native shared run-time library name information (for non-PIC native object files only)

  returns information on shared run-time library names (which are not necessarily the same as their file names) referenced by either the object file (*lib-info* is 0), the user library file (*lib-info* is 1), or the specified shared run-time library (*lib-info* is 2) in this variable-sized array.

  Returns a null list of SRLs if the file is PIC.

  Tal Value Representation **Description**

INT	Number of shared run-time library names returned. This value indicates how many pairs of INT and STRING listed, as below. follow this value.
INT	Length of the shared run-time library name.
STRING	Name. (The returned string ends on an odd-byte boundary so that the next attribute returned will begin on an even-byte boundary.)

Example

```
attr^list[0] := 12;  ! return compilation mode
attr^list[1] := 13;  ! return run mode
err := OBJFILE_GETINFOLIST_ (attr^list, 2,
                             return^values^list, 4,
                             return^values^len);
```

OLDFILENAME_TO_FILENAME_ Procedure

Summary

The OLDFILENAME_TO_FILENAME_ procedure converts a file name in the C-series internal file-name format to a file name in the D-series file-name format. See Appendix D, File Names and Process Identifiers for descriptions of C-series and D-series file names.

Syntax for C Programmers

```
#include <cextdecs(OLDFILENAME_TO_FILENAME_)>

short OLDFILENAME_TO_FILENAME_ ( short *oldfilename
                                ,char *filename
                                ,short maxlen
                                ,short *filename-length );
```

Syntax for TAL Programmers

```
error := OLDFILENAME_TO_FILENAME_ ( oldfilename          ! i
                                   ,filename:maxlen      ! o:i
                                   ,filename-length );   ! o
```

Parameters

error returned value

INT

is a file-system error number indicating the outcome of the operation.

oldfilename input

INT .EXT:ref:12

specifies a valid internal file name to be converted.

filename:*maxlen* output:input

STRING .EXT:ref:*, INT:value

contains the resulting file name. *maxlen* specifies the length in bytes of the string variable *filename*.

filename-length output

INT .EXT:ref:1

returns the actual byte length of the file name returned in *filename*. 0 is returned if an error occurs.

Considerations

- The output file name always includes a node name regardless of whether the input name was in network form. Note that the node name in the output is independent of the =_DEFAULTS DEFINE, as is the node name in the input. This is because the default node name in an internal file name is the node that the caller is running on, not the value in the =_DEFAULTS DEFINE.

- If the node number that is specified as part of *oldfilename* does not designate a node that is known to the local system, an *error* value of 18 is returned. In this case, the value returned in *filename* includes a printable string such as "\255," showing the node number that was supplied as input in place of a valid node name.
- When converting the process file name of a named or an unnamed process, OLDFILENAME_TO_FILENAME_ looks up the process in a system table and it might send a system message. An error 14 is returned if the process does not exist.

Related Programming Manual

For programming information about the OLDFILENAME_TO_FILENAME_ procedure, see the *Guardian Application Conversion Guide*.

OLDSYSMSG_TO_NEWSYSMSG_ Procedure

Summary

The OLDSYMSG_TO_NEWSYSMSG_ procedure converts a C-series format system message to its D-series equivalent. See "Considerations" for a list of the messages that this procedure accepts and produces.

Syntax for C Programmers

```
#include <cextdecs(OLDSYSMSG_TO_NEWSYSMSG_)>

short OLDSYSMSG_TO_NEWSYSMSG_ ( char *oldmsg
                               ,short length
                               ,char *newmsg
                               ,short maxlen
                               ,short *newmsg-length
                               ,[ short *error-detail ] );
```

Syntax for TAL Programmers

```
error := OLDSYSMSG_TO_NEWSYSMSG _ ( oldmsg:length        !
i:i
                                   ,newmsg:maxlen        !
o:i
                                   ,newmsg-length        ! o
                                   ,[ error-detail ] );  ! o
```

Parameters

error returned value

INT

indicates the result of the check. Valid values are:

0 Message successfully converted

1 File-system error; *error-detail* contains the file-system error number.

2 Parameter error; *error-detail* contains the number of the first parameter found to be in error, where 1 designates the first parameter on the left.

3 Bounds error; *error-detail* contains the number of the first parameter found to be in error, where 1 designates the first parameter on the left.

4 The supplied system message is not supported by this procedure; no conversion was performed.

oldmsg:*length* input:input

STRING .EXT:ref:*, INT:value

is the message to be converted. *oldmsg* must be exactly *length* bytes long.

newmsg:*maxlen* output:input

STRING .EXT:ref:*, INT:value

returns the equivalent D-series system message, if any. *maxlen* is the length in bytes of the string variable *newmsg*.

newmsg-length output

INT .EXT:ref:1

returns the actual length of the returned D-series system message, or 0 if the supplied message has no equivalent.

error-detail output

INT .EXT:ref:1

for some returned errors, contains additional information. See *error*, above.

Considerations

- A 250-byte buffer is adequate to hold any of the new messages. This value can always be used for the *maxlen* of *newmsg*.
- The old messages and new messages can be mapped one-to-one with the exception of the network status change message (-8). Depending on its content, the network status change message is converted into one of four new messages (see below).
- OLDSYSMSG_TO_NEWSYSMSG_ converts these system messages:

C-series message		D-series message	
-5	Process deletion: STOP	-101	Process deletion: STOP
-6	Process deletion: ABEND	-101	Process deletion: ABEND
-8	Network status change (all processors down)	-110	Loss of communication with node
-8	Network status change (single processor down, 0 or more processors up)	-100	Remote processor down
-8	Network status change (2 or more processors down, 0 or more processors up)	-110	Loss of communication with node
-8	Network status change (connection established)	-111	Establishment of communication with node
-8	Network status change (0 or more processors up when node already connected)	-113	Remote processor up
-9	Job process creation	-112	Job process creation

C-series message		D-series message	
-12	NEWPROCESSNOWAIT completion	-102	Nowait PROCESS_LAUNCH_ or PROCESS_CREATE_ completion
-20	Break on device	-105	Break on device
-30	Process open	-103	Process open
-31	Process close	-104	Process close
-40	Device type inquiry	-106	Device type inquiry

OPEN Procedure (Superseded by FILE_OPEN_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The OPEN procedure establishes a communication path between an application process and a file. When OPEN completes, a file number returns to the application

process. The file number identifies this access to the file in subsequent file-system calls.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
CALL OPEN ( file-name                              ! i
          ,filenum                                 ! o
          ,[ flags ]                               ! i
          ,[ sync-or-receive-depth ]               ! i
          ,[ primary-filenum ]                     ! i
          ,[ primary-process-id ]                  ! i
          ,[ seq-block-buffer-id ]                 ! i
          ,[ buffer-length ]                       ! i
          ,[ primary-define ] );                   ! i
```

Parameters

file-name input

INT:ref:12

is an array containing the internal-format file name of the file to be opened. For additional information about file names, see Appendix D, File Names and Process Identifiers .

Note that *file-name* can be a DEFINE name. For additional information about DEFINEs, see Appendix E, DEFINEs.

filenum output

INT:ref:1

returns a number used to identify the file in subsequent system calls. A -1 is returned if OPEN fails.

flags input

INT:value

specifies certain attributes of the file. If omitted, all fields are set to 0. The bit fields in the *flags* parameter are defined in Table 11-2.

sync-or-receive-depth input

INT:value

The purpose of this parameter depends on the type of device being opened:

Disk file specifies the number of nonretryable (that is, write) requests whose completion the file system must remember. A value of one or greater must be specified to recover from a path failure occurring during a write operation. This value also implies the number of write operations the primary process in a primary and backup process pair can perform to this file without intervening checkpoints to its backup process. For disk files, this parameter is called sync depth. The maximum sync depth value is 15.

If omitted, or if 0 is specified, internal checkpointing does not occur. Disk path failures are not automatically retried by the file system.

$RECEIVE file
specifies the maximum number of incoming messages read by READUPDATE that the application process is allowed to queue before corresponding REPLYs must be performed.

If omitted, READUPDATE and REPLY to $RECEIVE are not permitted.

For $RECEIVE, this parameter is called receive-depth, and the maximum number of queued incoming messages is 4047 in the H06.17/J06.06 and earlier RVUs. From H06.18/J06.07 RVU onwards, the maximum receive-depth value has been increased from 4047 to 16300.

process pair
specifies whether or not an I/O operation is automatically redirected to the backup process if the primary process or its processor module fails. For processes, this parameter is called sync depth. The maximum value is determined by the process. The value must be at least 1 for an I/O operation to a remote process pair to recover from a network failure.

If this parameter >= 1, the server is expected to save or be able to regenerate that number of replies.

If this parameter = 0, and if an I/O operation cannot be performed to the primary process of a process pair, an error indication is returned to the originator of the message. On a subsequent I/O operation, the file system redirects the request to the backup process.

For other device types, the meaning of this parameter depends on whether the sync-ID mechanism is supported by the device being opened. If the device does not support the sync-ID mechanism, 0 is used regardless of what you specify (this is the most common case). If the device supports the sync-ID mechanism, specifying a nonzero value causes the results of that number of operations to be saved; in case of failures, the operations can be retried if necessary.

The actual value being used can be obtained by a call to FILE_GETINFOLIST_ or FILEINFO.

primary-filenum input

INT:value

is the file number returned to the primary process when it opened this file. *primary-filenum* must be passed as -*filenum*.

primary-filenum and *primary-process-id* are supplied *only* if the open is by the backup process of a process pair, the file is currently open by the primary process, *and* the checkpointing facility is *not* used. Both parameters must be supplied.

A negative file number indicates that the same file number should be returned in the backup as was returned in the primary. If a negative file number is specified and the file number is already open by the backup process, OPEN returns file-system error 12. In this situation, a process pair would indicate externally that error 12 (file in use) exists when, in fact, the file is not in use by the normal definition (open by another process in exclusive mode).

primary-process-id input

INT:ref:4

is an array that contains the 4-word *process-id* of the corresponding primary process.

primary-process-id and *primary-filenum* are supplied *only* if the open is by the backup process of a process pair, the file is currently open by the primary process, *and* the checkpointing facility is *not* used. Both parameters must be supplied.

seq-block-buffer-id input

INT:ref:1

is a 16-bit value, the address of which identifies the sequential block buffer to be shared, if sequential block buffering is used and if sharing is desired. If sharing is not desired, this parameter can be omitted since all sequential buffers will reside in the process's PFS with the size given by *buffer-length*. All opens giving this

ID share the same sequential block buffer. Any integer value can be supplied for this parameter.

If sequential block buffering is used, the file should usually be opened with protected or exclusive access. Shared access can be used, although there are potential concurrency problems, and it is somewhat slower than other access methods in the case of key-sequenced files. See the discussion of “Sequential Block Buffering” in the *Enscribe Programmer’s Guide*.

buffer-length input

INT:value

is the length in bytes of the sequential block buffer. This is the only parameter that is required for the sequential block buffering option to be in effect when buffer sharing is not used.

If the *buffer-length* is less than the *data-blocklen* specified in the creation of this file or any associated alternate-key files, then the larger size is used, unless a buffer that was established by an earlier call to OPEN is being shared and is too small. In that case OPEN succeeds but returns a CCG indication (a subsequent call to FILEINFO or FILE_GETINFO_ shows that an error 5 occurred). Normal system buffering is then used instead of the application process’s sequential buffer.

If this parameter is omitted, sequential block buffering is not attempted.

primary-define input

INT:ref:12

specifies the name of the DEFINE which was used as the *file-name* in the open of the primary process. (In the backup, *file-name* must be the actual name of the file.) The DEFINE must exist and must have the same value as it did when the primary open was made. This parameter is relevant only for a process pair which does not use the CHECKOPEN or CHECKMONITOR procedures.

The *primary-define* parameter should be supplied only if this open is a backup open and the primary open was made using a DEFINE.

Condition Code Settings

< (CCL) indicates that the open failed (call FILEINFO or FILE_GETINFO_). If OPEN fails, a -1 is returned in *filenum*.= (CCE)indicates that the file was opened successfully.

> (CCG) indicates the file was opened successfully but an exceptional condition was detected (call FILEINFO or FILE_GETINFO_).

Table 11-2. OPEN flags Parameter (page 1 of 2)

Flag	Flag in Octal	Meaning
<0>	%100000	For disk files, if this bit is 1, the “last open time” attribute of the file being opened is not updated by this open. For other files, this bit should be zero.
<1>	%40000	For the $RECEIVE file only, specifies whether the opener wants to receive open, close, CONTROL, SETMODE, SETPARAM, RESETSYNC, and CONTROLBUF system messages. Note that some messages are received only with SETMODE 80. 0 = no, 1 = yes (must be 0 for all files other than $RECEIVE)
<2>	%20000	Specifies that access to an Enscribe file is to occur as if the file were unstructured, that is, without regard to record structures and partitioning, (Note that for unstructured files, setting this bit to 1 makes secondary partitions inaccessible.) Setting this bit to 0 provides normal structured access to the file. 0 = normal access, 1 = unstructured access
<3>	%10000	(Reserved) must be 0 for nonprivileged users.
<4:5>	%6000 (If both bits set)	Access mode 0 = Read/write 1 = Read-only 2 = Write-only 3 = reserved
<6>	%1000	Must be 0 (reserved)
<7>	%400	Must be 0 (reserved)
<8>	%200	For process files, indicates that the open message is sent nowait and must be completed with a call to AWAITIO[X]. OPEN returns a valid file number. 0 = no, 1 = yes (must be 0 for all other files)

Table 11-2. OPEN flags Parameter (page 2 of 2)

Flag	Flag in Octal	Meaning
<9>	%100	Must be 0 (reserved)
<10:11>	%60 (If both bits set)	Exclusion mode 0 = shared 1 = exclusive 2 = process exclusive (supported for Optical Storage Facility only) 3 = protected
<12:15>	%17 (If all four bits set)	> 0 implies nowait I/O and the maximum number of concurrent nowait I/O operations that can be in progress on this file at any given time. 0 implies waited I/O.

Considerations

- File numbers

  Within a process, the file numbers are unique. The lowest numeric file number is 0 and is reserved for $RECEIVE. Remaining file numbers start at 1. The lowest available file number is always assigned. Once a file is closed, its file number becomes available, and a subsequent open can reuse that file number.

- Maximum number of open files

  The maximum number of files in the system that can be open at any given time depends on the space available for control blocks; access control blocks (ACBs), file control blocks (FCBs), and open control blocks (OCBs). The amount of space available for control blocks is limited primarily by the physical memory size of the system. Each process can have up to one megabyte of space for ACBs; the default is 128 kilobytes for ACBs.

- Multiple openings by the same process

  If a given file is opened more than once by the same process, a new ACB is created for each open. This provides logically separate accesses to the same file because a unique file number returns to the process for each open. Whenever you reference a file in a procedure, the file number is supplied by you in the *filenum* parameter of the procedure.

  Multiple opens on a given file can create a deadlock. This shows how a deadlock situation occurs:

```
OPEN( MYFILE , filenuma ... );
! first open on file MYFILE.
      .
      .
OPEN( MYFILE , filenumb ... );
```

```
! second open on file MYFILE.
     .
     .
OPEN( MYFILE , filenumc ... );
! third open on file MYFILE.
     .
----
d       .
e  LOCKFILE ( filenumb, ... );     ! the file is locked
a       .                          ! using the file number
d       .                          ! associated with the
l       .                          ! second open.
o  READUPDATE ( filenumc, ... );   ! update the file
c       .                          ! associated with the
k       .                          ! third open.
----
```

Locks are granted on an open file (that is, file number) basis. Therefore, if a process has multiple opens of the same file, a lock of one file number excludes access to the file through other file numbers. The process is suspended forever if the default locking mode is in effect.

You now have a deadlock. The file number referenced in the LOCKFILE call differs from the file number in the READUPDATE call.

- Limit number of times file can be open

  There is a limit to the total number of times a given file can be open at one time. This determination includes opens by all processes.

  The specific limit for a file is dependent on the file's device type:

Disk Files	Cannot exceed 32,767 opens per disk
Process	Defined by process (see discussion of "controlling openers" in the *Guardian Programmer's Guide*)
$0	Unlimited opens
$0.#ZSPI	128 concurrent opens permitted
$OSP	10 times the number of subdevices (up to a maximum of 83 subdevices)
$RECEIVE	One open per process permitted
Other	Varies by subsystem

- Nowait opens—errors

  If a process file is opened in a nowait manner (*flags*.<8> = 1), that file is opened as nowait and checkopened in a nowait manner. Errors detected in parameter specification and system data space allocation are returned by the call to OPEN, and the operation is considered unsuccessful. If there is an error, no message to the process being opened is sent, and no call to AWAITIO is needed to complete the open.

If there are no parameter or data space allocation errors, the *filenum* parameter is valid when OPEN returns. However, no I/O operation on the file can be initiated until the open is completed, and other errors are reported by a call to AWAITIO.

If the *tag* parameter is specified in the call to AWAITIO, a -30D returns. The values returned in the buffer and count parameters to AWAITIO are undefined. If an error returns from AWAITIO, it is the user's responsibility to close the file.

For a nonprocess or waited (nowait depth = 0) file, *flags*.<8> is internally reset to 0 and ignored. A call to FILEINFO after the call to OPEN can return the value of the internal flags; if bit <8> = 1, then a call to AWAITIO must be performed to complete the open.

For considerations when using nowait I/O, see the *Enscribe Programmer's Guide*. For a general discussion of nowait I/O, see the *Guardian Programmer's Guide*.

- Direct and buffered I/O transfers

  A file opened by OPEN uses an intermediate buffer in the process file segment (PFS) for I/O (read) transfers by default; SETMODE 72 is used to force the system to use direct I/O transfers. This is unlike FILE_OPEN_, which uses direct I/O transfers by default.

  The system buffers are used for files opened by OPEN. If you want to use user buffers instead of system buffers, set SETMODE 72,2. Note that calling the USERIOBUFFER_ALLOW_ procedure before the OPEN procedure does not override the implicit SETMODE 72,1 for files opened by OPEN.

- Partitioned files

  A separate pair of FCBs exist for each partition of a partitioned file. There is one ACB per accessor (as for single-volume files), but this ACB requires more main memory since it contains the information necessary to access all of the partitions, including the location, alternate keys, and partial-key value for each partition.

- Disk file open—security check

  When a disk file open is attempted, the system performs a security check. The accessor's (that is, the caller's) security level is checked against the file security level for the requested access mode, as follows:

for read access:	read security level is checked.
for write access:	write security level is checked.
for read-write access:	read and write security levels are checked.

  A file has one of seven levels of security for each access mode. (The owner of the file can set the security level for each access mode by using SETMODE function 1 or by using the File Utility Program SECURE command.) Table 11-3 shows the seven levels of security.

Table 11-3. Levels of Security

FUP Code	Program Values	Access
–	7	Local super ID only
U	6	Owner (local or remote), that is, any user with owner's ID
C	5	Member of owner's group (local or remote), that is, any member of owner's community
N	4	Any user (local or remote)
O	2	Owner only (local)
G	1	Member of owner's group (local)
A	0	Any user (local)

For a given access mode, the accessor's security level is checked against the file security level. File access is allowed or not allowed as shown in Table 11-4. In this table, file security levels are indicated by FUP security codes. For a given accessor security level, a Y indicates that access is allowed to a file with the security level shown; a hyphen indicates that access is not allowed.

Table 11-4. Allowed File Accesses

Accessor's Security Level	File Security Level							
	–	U	C	N	O	G	A	
Super ID user, local access	Y	Y	Y	Y	Y	Y	Y	
Super ID user, remote access	–	Y	Y	Y	–	–	–	
Owner or owner's group manager, remote access	–	Y	Y	Y	–	–	–	
Member of owner's group, remote access	–	–	Y	Y	–	–	–	
Any other user, remote access	–	–	–	Y	–	–	–	
Owner or owner's group manager, local access	–	Y	Y	Y	Y	Y	Y	
Member of owner's group, local access	–	–	Y	Y	–	Y	Y	
Any other user, local access	–	–	–	Y	–	–	Y	

If the caller to FILE_OPEN_ fails the security check, the open fails with an error 48. A file's security can be obtained by a call to FILE_GETINFOLIST[BYNAME]_ , FILEINFO, or by the File Utility Program (FUP) INFO command.

If you are using the Safeguard product, this security information might not apply.

- Tape file open—access mode

The file system does not enforce read-only or write-only access for unlabeled tape, even though no error is returned if you specify one of these access modes when opening a tape file.

- File open—exclusion and access mode checking

  When a file open is attempted, the requested access and exclusion modes are compared with those of any opens already granted for the file. If the attempted open is in conflict with other opens, the open fails with error 12.

Table 11-5 lists the possible current modes and requested modes, indicating whether an open succeeds or fails.

Note. Protected exclusion mode has meaning only for disk files. For other files, specifying protected exclusion mode is equivalent to specifying shared exclusion mode.

Table 11-5. Exclusion and Access Mode Checking

Exclusion Mode		File currently open with:									File Closed
		Shared			Exclusive			Protected			
Open attempted with:	Access Mode	Read/ Write	Read Only	Write Only	Read/ Write	Read Only	Write Only	Read/ Write	Read Only	Write Only	
Shared	Read/ Write	●	●	●							●
	Read Only	●	●	●				●	●	●	●
	Write Only	●	●	●							●
Exclusive	Read/ Write	Always Fails									●
	Read Only										●
	Write Only										●
Protected	Read/ Write		●								●
	Read Only		●						●		●
	Write Only		●								●

Legend

● = Open Successful

□ = Open Fails

Note: When a program file is running, it is opened with the equivalent of protected, read-only access.

CDT 008.CDD

- Applications with large receive-depth values

  If you have applications that use large receive-depth values, you must periodically monitor their Message Quick Cell (MQC) usage levels using the `PEEK /CPU N/ MQCINFO` command in all the processors to make sure that the total amount of memory allocated for MQCs does not approach the per-processor memory limit for

MQCs. This limit is 128 MB in H06.19 / J06.08 and earlier RVUs, and 1 GB in H06.20 / J06.09 and later RVUs. For more information, see Table J-3 on page 3.

If you run applications with very large receive-depth values on systems running H06.19/J06.08 or earlier RVUs, you must consider upgrading to H06.20/J06.09 or a later RVU if you notice MQC memory usage levels approach the per-processor memory limit of 128 MB.To determine the amount of memory used for MQCs by CPU N from the `PEEK /CPU N/ MQCINFO` command output, add the page counts for all MQC sizes, and then multiply the total page count allocated for MQCs by the page size (16 KB).

Disk File Considerations

- Maximum number of concurrent nowait operations

  The maximum number of concurrent nowait operations permitted for an open of a disk file is one. Attempting to open a disk file and specify a value greater than 1 returns an error indication. A subsequent call to FILEINFO or FILE_GETINFO_ shows that an error 28 occurred.

- Unstructured files

  - File pointers after open

    After a disk file is opened, the current-record and next-record pointers begin at a relative byte address (RBA) of zero, and the first data transfer (unless an intervening POSITION is performed) is from that location. After a successful open, the pointers are:

    current-record pointer = 0D
    next-record pointer = 0D

  - Sharing the same EOF pointer

    If a given disk file is opened more than once by the same process, separate current-record and next-record pointers are provided for each open, but all opens share the same EOF pointer.

- Structured files

  - Accessing structured files as unstructured files

    The unstructured access option (*flags*.<2>) permits a file to be accessed as an unstructured file. For OPEN, with this option specified, a data transfer occurs to the position in the file specified by an RBA (instead of to the position indicated by a key address field or record number); the number of bytes transferred is that specified in the file-system procedure call (instead of the number of bytes indicated by the record format). If a partitioned, structured file is opened as an unstructured file, only the first partition is opened. The remaining partitions must be opened individually with separate calls to OPEN (each call to OPEN specifying unstructured access).

    Accessing audited structured files as unstructured files is not allowed.

△ **Caution.** Programmers using this option are cautioned that the block format used by Enscribe must be maintained if the file is to be accessed again in its structured form. (HP reserves the right to change this block format at any time.) For information about Enscribe block formats, see the *Enscribe Programmer's Guide*.

- Current-state indicators after open

  After successful completion of OPEN, the current-state indicators have these values:

  - The current position is that of the first record in the file by primary key.

- The positioning mode is approximate.
- The comparison length is 0.

  If READ is called immediately after OPEN for any structured file, it reads the first record in the file; in a key-sequenced file, this is the first record by primary key. Subsequent reads, without intervening positioning, read the file sequentially or by primary key through the last record in the file.

  When a key-sequenced file is opened, KEYPOSITION usually is called before any subsequent I/O call (such as READ, READUPDATE, WRITE) to establish a position in the file.

- Queue files

  If the READUPDATELOCK[X] operation is to be used, the *sync-or-receive-depth* parameter must be 0. A separate open may be used for operations with *sync-or-receive-depth* > 0.

  Sequential block buffering cannot be used.

Terminal Consideration

The terminal being used as the operator console should not be opened with exclusive access. If it is, console messages are not logged.

Interprocess Communication Considerations

- Maximum concurrent nowait operations for an open of $RECEIVE

  The maximum number of concurrent nowait operations permitted for an open of $RECEIVE is one. Attempting to open $RECEIVE and to specify a value greater than 1 returns an error indication. A subsequent call to FILEINFO or FILE_GETINFO_ shows that an error 28 occurred.

- When open completes

  When process A attempts to open process B, open completes as follows:

  - The open for process A won't complete if process B has not opened its $RECEIVE.
  - If process B has opened its $RECEIVE, but has not requested system messages, the open for process A completes immediately.
  - If process B has opened its $RECEIVE requesting system messages, and with a *receive-depth* equal to 0, the open for process A completes when process B does a read of $RECEIVE to get the open message from A.
  - If process B has opened its $RECEIVE requesting system messages and with *receive-depth* greater than 0, the open for process A completes after process B has read the open message of process A and replied to it.

- Opening high-PIN processes

  The OPEN procedure cannot be used to open a high-PIN unnamed process because the process ID cannot fit into the process file name; FILE_OPEN_ must be used instead. However the OPEN procedure can be used on high-PIN named processes, devices, and files on high-PIN volumes.

- Opening an unconverted (C-series) process from a high-PIN process.

  A high-PIN process cannot open an unconverted process unless the unconverted process has the HIGHREQUESTERS object-file attribute set. If a high-PIN process attempts to open a low-PIN process that does not have this attribute set, file-system error 560 occurs.

- Opening $RECEIVE and being opened by a remote long-named process

  If a process uses the OPEN procedure to open $RECEIVE (or if it uses the FILE_OPEN_ procedure to open $RECEIVE and requests that C-series format messages be delivered), then a subsequent open of that process (using either OPEN or FILE_OPEN_) by another process on a remote node that has a process name consisting of more than five characters will fail with an error 20.

Message

The process open system message is received by a process when it is opened by another process. The 4-word process ID of the opener can be obtained in a subsequent call to FILE_GETRECEIVEINFO_ , LASTRECEIVE, or RECEIVEINFO. For a description and the form of this message and all system messages, see the *Guardian Procedure Errors and Messages Manual*.

Note. This message is also received if the backup process of a process pair performs the open. Therefore, a process can expect two of these messages when being opened by a process pair.

DEFINE Considerations

The *file-name* parameter can be a DEFINE name; OPEN will use the file name given by the DEFINE as the object to be opened. If a CLASS TAPE DEFINE without the DEVICE attribute is referenced, a specific tape drive will be selected. A DEFINE of CLASS TAPE has other effects when supplied to OPEN; see Appendix E, DEFINEs for further information about DEFINEs.

If no DEFINE exists for the specified DEFINE name, the procedure returns error 198 (missing DEFINE).

Safeguard Considerations

For information on files protected by Safeguard, see the *Safeguard Reference Manual*.

OSS Considerations

- This procedure operates only on Guardian objects. If an OSS file is specified, error 1163 is returned.

Example

```
CALL OPEN ( FILE^NAME , FILE^NUM );
```

The file in this call has these defaults; wait I/O, exclusion mode (shared), access mode (read/write), sync depth (0).

Related Programming Manuals

For programming information about the OPEN file-system procedure, see the *Enscribe Programmer's Guide*.

OPEN^FILE Procedure

Summary

The OPEN^FILE procedure permits access to a file when using sequential I/O (SIO) procedures.

Syntax for C Programmers

```
#include <cextdecs(OPEN_FILE)>

short OPEN_FILE ( short _near *common-fcb
                 ,short _near *file-fcb
                 ,[ short _near *block-buffer ]
                 ,[ short block-bufferlen ]
                 ,[ __int32_t flags ]
                 ,[ __int32_t flags-mask ]
                 ,[ short max-recordlen ]
                 ,[ short prompt-char ]
                 ,[ short _near *error-file-fcb ] );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
error := OPEN^FILE ( common-fcb                    ! i
                    ,file-fcb                      ! i
                    ,[ block-buffer ]              ! i
                    ,[ block-bufferlen ]           ! i
                    ,[ flags ]                     ! i
                    ,[ flags-mask ]                ! i
                    ,[ max-recordlen ]             ! i
                    ,[ prompt-char ]               ! i
                    ,[ error-file-fcb ] );         ! i
```

Parameters

error returned value

INT

returns a file-system or SIO procedure error number indicating the outcome of the operation.

If the abort-on-open-error mode is in effect (the default situation), the only possible value of *error* is 0.

common-fcb input

INT:ref:*

is an array of FCBSIZE or FCBSIZE^D00 words for use by the SIO procedures. Only one common file control block (FCB) is used per process. This means the same data block is passed to all OPEN^FILE calls. The common FCB must be initialized before the first call to OPEN^FILE following a process startup. The size of the file control block differs between TNS processes and native processes.

file-fcb input

INT:ref:*

is an array of FCBSIZE or FCBSIZE^D00 words for use by the SIO procedures. The file FCB uniquely identifies this file to other SIO procedures. The file FCB must be initialized with the name of the file to be opened before OPEN^FILE is called. The size of the file control block differs between TNS processes and native processes.

For information about the FCB structure, see the *Guardian Programmer's Guide*.

block-buffer input

INT:ref:*

is an array used for one of four different purposes:

- When reading a structured file, the presence of this parameter indicates a request for sequential block buffering. If more than one file refers to the same block-buffer address, they share the same sequential block buffer.
- When reading or writing an EDIT file, the buffer is used by SIO to contain EDIT file pages being assembled or disassembled. The buffer must be supplied for an EDIT file.
- When using level-3 spooling, the buffer is used by SIO to hold records that are to be sent to a spooler collector.
- If *block-buffer* is not being used for any of the other three purposes, then the array is used for SIO record blocking and deblocking. No blocking is performed if any of these occurs:
  - *block-buffer* or *block-bufferlen* is omitted.
  - The value of *block-bufferlen* is insufficient according to the record length for the file.
  - Read/write access is indicated.

  Blocking occurs when *block-buffer* is supplied, the block buffer is of sufficient length (as indicated by *block-bufferlen*), and blocking is appropriate for the device.

  For TNS processes, the block buffer must be located within 'G'[0:32767] of the data area. This limit does not apply to native processes.

block-bufferlen input

INT:value

indicates the length, in bytes, of the block buffer. This length must be able to contain at least one logical record. For an EDIT file, the minimum length on read is 144 bytes; on write, the minimum length is 1024 bytes. For use with level-3 spooling, the minimum length is 1024 bytes.

flags input

INT(32):value

if present, is used with the *flags-mask* parameter to set file transfer characteristics. If omitted, all positions are treated as zeros.

These literals can be combined using signed addition because bit 0 is not used:

ABORT^OPENERR	KEEP^LASTOPENTIME	PURGE^DATA
ABORT^XFERERR	LEVEL3^SPOOL^ENABLE	READ^TRIM
AUTO^CREATE	MUSTBENEW	VAR^FORMAT
AUTO^TOF	NOWAIT	WRITE^FOLD
BLOCKED	OLD^RECEIVE	WRITE^PAD
CRLF^BREAK	PRINT^ERR^MSG	WRITE^TRIM

For the meanings of literals used with *flags*, see General Considerations on page 11-33. Literal declarations are contained in the file $SYSTEM.SYSTEM.GPLDEFS.

flags-mask input

INT(32):value

specifies which bits of the flag field are used to alter the file transfer characteristics. A characteristic to be altered is indicated by entering a 1 in the bit position corresponding to the *flags* parameter. A 0 indicates the default setting is used. If this parameter is omitted, all positions are treated as zeros.

max-recordlen input

INT:value

specifies the maximum record length for records within this file. If this parameter is omitted, the maximum record length is 132.

The open is aborted with an SIOERR^INVALIDRECLENGTH, error 520, if the file's record length exceeds the maximum record length and *max-recordlen* is not 0. If *max-recordlen* is 0, then any record length is permitted.

prompt-char input

INT:value

specifies the interactive prompt character for reading from terminals or processes. “?” is the default prompt. The prompt character is limited to seven bits, <9:15>.

error-file-fcb input

INT:ref:*

if present, specifies a file where error messages are displayed for all files. Only one error-reporting file is allowed per process. The file specified in the latest OPEN^FILE call is the one used. Omitting this parameter does not alter the setting of the current error-reporting file.

The default error-reporting file is the home terminal.

If the error-reporting file is not open when needed, it is opened only for the duration of the message printing, then closed. Remember that the error-reporting file FCB must be initialized.

For information about the file FCB, see the *Guardian Programmer’s Guide*.

General Considerations

- Specifics of AUTO^TOF

  If AUTO^TOF is ON, a top-of-form control operation is performed to the file when both (1) the file being opened is a process or a line printer, and (2) write access is specified.

- When read/write access is not permitted

  If the file is an EDIT file or if blocking is specified, either read or write access must be specified for the open to succeed. Read/write access is not permitted.

- Accessing a temporary disk file

  When using OPEN^FILE to access a temporary disk file, AUTO^CREATE must be OFF; otherwise, the OPEN^FILE call results in a file-system error 13.

- Sync depth of open files

  All files opened with the OPEN^FILE procedure are opened with a sync depth of 1. This is the only possible sync depth; no other can be set.

- Opening $RECEIVE

  If you attempt to use a C-series format common FCB with a D-series format FCB for $RECEIVE, OPEN^FILE fails with an error 536.

- Error-reporting file

The error-reporting file is used, when possible, for reporting errors. If this file cannot be used or if the error is with the error-reporting file, the default error-reporting file is used.

- Appending data to the file

  SIO procedures append data to the file if access is write only and PURGE^DATA is OFF (the default value).

- Opening $RECEIVE with the OLD^RECEIVE flag ON

  If $RECEIVE is open with the OLD^RECEIVE flag ON (receive C-series format messages), a subsequent open of the caller by another process on a remote node that has a name consisting of more than five characters fails with an error 20. Notification of this failure is not sent to the caller reading $RECEIVE.

- List of literals used with *flags* and *flags-mask*

ABORT^OPENERR	(%1D) Abort on open error; defaults to ON (1). If ON and a fatal error occurs during the OPEN^FILE call, all files are closed and the process ends abnormally. If OFF (0), the file-system or SIO-procedure error number is returned to the caller.
ABORT^XFERERR	(%2D) Abort on data transfer error; defaults to ON. If ON and a fatal error occurs during a data transfer operation (such as a call to any SIO procedure except OPEN^FILE), all files are closed and the process ends abnormally. If OFF, the file-system or SIO-procedure error number is returned to the caller.
AUTO^CREATE	(%10D) Auto create; defaults to ON. If ON, and if open access is write, a file is created if one does not already exist. If write access is not given and the file does not exist, error 11 is returned. If no file code has been assigned or if the file code is 101, and if a block buffer of at least 1024 bytes is provided, an EDIT file is created. If there is not a buffer of sufficient size and no new file code is specified, then a file code of 0 is used. (See EDIT File Considerations on page 11-36.) The default extent sizes are 8 pages for the primary extent and 32 pages for the secondary extent. The maximum number of extents is 500.
AUTO^TOF	(%100D) Auto top of form; defaults to ON. If ON and the file is a line printer or process that is open with write access, a page eject is issued to the file within the OPEN^FILE procedure.
BLOCKED	(%400D) Nondisk blocking; defaults to OFF. A block buffer of sufficient length must also be specified.

CRLF^BREAK (%40000D) Carriage return/line feed (CR/LF) on BREAK; defaults to ON. If ON and BREAK is enabled, a CR/LF is written to the terminal when BREAK is entered.

KEEP^LASTOPENTIME
(%400000D) Keep last open time; defaults to OFF. If ON and open access to a disk file is read only, the “time of last open” file attribute is not updated by this open. If OFF, the “time of last open” file attribute is updated. This flag is ignored if the file is not a disk file or if open access is not read only.

LEVEL3^SPOOL^ENABLE
(%200000D) Enable level-3 spooling when writing to a spooler collector; defaults to OFF. If ON, writing to the spooler collector is buffered and a block buffer with a length of at least 1024 bytes must be provided. If OFF or if the other requirements for level-3 spooling are not met, one record at a time is written to the spooler collector. See Level-3 Spooling Considerations on page 11-37.

MUSTBENEW (%20D) File must be new; defaults to OFF. This flag applies only if AUTO^CREATE is ON. If the file already exists, error 10 is returned.

NOWAIT (%200D) Nowait I/O; defaults to OFF (wait I/O). If ON, nowait I/O is in effect. If NOWAIT is specified in the open flags of OPEN^FILE, then the nowait depth is 1. It is not possible to use a nowait depth greater than 1 using SIO procedures.

OLD^RECEIVE (%100000D) Receive C-series format system messages; defaults to OFF. If ON, system messages read by the caller from $RECEIVE are in C-series format. If OFF, system messages read from $RECEIVE are in D-series format. This flag is ignored for all files other than $RECEIVE. It is also ignored if you are using a C-series format FCB; all messages are then in C-series format.

PRINT^ERR^MSG (%4D) Print error message; defaults to ON. If ON, and a fatal error occurs, an error message is displayed on the error file. This file is the home terminal unless otherwise specified.

PURGE^DATA (%40D) Purge data; defaults to OFF. If ON, and open access is write, the data is purged from the file after the open. If OFF, the new data is appended to the existing data.

READ^TRIM	(%2000D) Read trailing blank trim; defaults to ON. If ON, the *count-returned* parameter on a READ^FILE call does not account for trailing blanks.
VAR^FORMAT	(%1000D) Variable-length records; defaults to OFF for fixed-length records. If ON, the maximum record length for variable-length records is 254 bytes.
WRITE^FOLD	(%10000D) Write fold; defaults to ON. If ON, writes that exceed the record length cause multiple logical records to be written. If OFF, writes that exceed the record length are truncated to record-length bytes; no error message or warning is given.
WRITE^PAD	(%20000D) Write blank pad; defaults to ON for disk fixed-length records and OFF for all other files. If ON, writes of less than record-length bytes, including the last record if WRITE^FOLD is in effect, are padded with trailing blanks to fill out the logical record.
WRITE^TRIM	(%4000D) Write trailing blank trim; defaults to ON. If ON, trailing blanks are trimmed from the output record before being written to the file.

EDIT File Considerations

- When creating a file, if you do not assign a file code for the new file or if you assign it a file code of 101, and if you provide a block buffer of at least 1024 bytes, an EDIT file is created. If you do not provide a block buffer of sufficient size and if you assign no file code, a file code of 0 is used. If you assign a file code of 101 but do not provide a block buffer of sufficient size, an error is returned.
- EDIT files are created with the ODDUNSTR attribute set. EDIT files created before the D20 RVU might not have this attribute set. When opening an EDIT file created before D20 that does not have the ODDUNSTR attribute set, SIO alters the file so that it has this attribute set.
- Starting in the D20 RVU, the EDIT file directory is written to the end of a new disk extent whenever a new extent is used (although in the case of a new EDIT file, the directory is not moved to the end of the primary extent until the first page is filled). The result is that, starting in D20, an EDIT file might appear to be larger (that is, the value of EOF is greater) when compared to the same file managed by SIO before D20. The file is not actually larger, as the same amount of disk space is allocated in both cases. The difference is that in the newer version, the last part of each new extent is immediately put into use.

- SIO always performs buffered I/O, using the disk process buffering mechanism to enhance performance, when writing to an EDIT file. You can force SIO to flush buffered data to disk by calling WRITE^FILE and specifying a *write-count* of -1.
- For performance reasons, it is recommended that you provide a block buffer with a length of about 2100 bytes. This is because SIO normally requires slightly more than 2048 bytes to assemble EDIT file pages.
- You must specify either read or write access when opening an EDIT file; read/write access is not permitted.

Level-3 Spooling Considerations

- Level-3 spooling allows multiple records to be sent per message to a spooler collector, greatly reducing the number of messages required to do spooling. To use level-3 spooling with SIO, you must open a spooler collector by calling OPEN^FILE. These requirements must be met:
  - You must set the LEVEL3^SPOOL^ENABLE flag ON in the call to OPEN^FILE.
  - You must provide a block buffer with a length of at least 1024 bytes.
  - The open exclusion mode of the file must be shared.
  - The maximum record length of the print line buffer is 900 bytes.

  If any of these requirements are not met, level-3 spooling is not enabled. You can verify whether level-3 spooling is enabled by calling the CHECK^FILE procedure and specifying the FILE^LEVEL3^SPOOLING operation.
- CONTROL or SETMODE operations are not allowed on a file that is opened by SIO for level-3 spooling; error 2 is returned by CONTROL or SETMODE for any operation. Certain CONTROL operations can be requested in a call to OPEN^FILE or WRITE^FILE; these continue to be available when a file is opened for level-3 spooling.
- The spooler interface procedures, through which SIO performs spooling, do not support nowait I/O. You can set the NOWAIT flag ON in a call to OPEN^FILE and WRITE^FILE, but SIO still performs I/O operations to a spooler collector in a waited manner.

Example

```
ERROR := OPEN^FILE ( COMMON^FCB , IN^FILE , BUFFER
             , BUFFER^SIZE , FLAGS , FLAGS^MASK , , PROMPT );
```

Related Programming Manual

For programming information about the OPEN^FILE procedure, see the *Guardian Programmer's Guide*.

OPENEDIT Procedure (Superseded by OPENEDIT_ Procedure)

Summary

Note. The OPENEDIT procedure is supported for compatibility with previous software. For new development, the OPENEDIT_ procedure should be used instead.

The OPENEDIT procedure allocates and initializes data blocks in the EDIT file segment (EFS) so that the specified file can be accessed later by the IOEdit procedures. It optionally creates and opens the specified file through the file system.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
error := OPENEDIT ( file-name                    ! i
                  ,filenum                       ! i,o
                  ,[ flags ]                     ! i
                  ,[ sync-depth ]                ! i
                  ,[ write-thru ] );             ! i
```

Parameters

error returned value

INT

returns a file-system error number indicating the outcome of the operation. The possible values include:

-1 Page-count value is inconsistent.
-2 Page-table tags are out of order.
-3 Page-table tag is outside valid range.
-4 Page-table block number is outside of file.
-5 Page table has duplicate block numbers.

11 File does not exist; indicates that the file does not exist and that the *flags* parameter indicates read-only access to the file.

14 Device does not exist; indicates that the device-name part of the file name designates a device that either does not exist or is not a disk device.

16 File has not been opened, wrong file type; indicates that the file is not an EDIT file (that is, the file type is not unstructured or the file code is not 101 or 102).

31 Unable to obtain buffer space; indicates that the file's directory does not fit into IOEdit's extended data segment and OPENEDIT is unable to enlarge the segment.

34 Unable to obtain memory space for control block; indicates that the number of IOEdit files already open is equal to the maximum number specified or assumed when INITIALIZEEDIT was called.

59 File is bad; indicates that the file exists and has the correct file type and file code for an EDIT file, but the data in the file has an incorrect format and OPENEDIT is unable to repair it.

The negative values listed above indicate that OPENEDIT has found a format error in the file that it can probably correct. At the time that one of these values is returned, the file has not yet been altered. If you immediately call CLOSEEDIT or CLOSEEDIT_, the file is closed without change. If instead you call another IOEdit procedure, IOEdit tries to correct the format of the file. The format corrections are written to disk when CLOSEEDIT or CLOSEEDIT_ is finally called. If IOEdit fails to correct the format error, error 59 is returned for all subsequent IOEdit operations on the file.

file-name input

INT .EXT:ref:12

is an array containing the internal-format file name that identifies the file to be opened. If the file must be created, this name is assigned to it.

filenum input, output

INT:value

on input, if the specified value is greater than or equal to 0 and if the file designated by *file-name* exists, indicates that the caller has already opened the file through the file system and that *filenum* is the number returned by the file system to identify the open file. In this case, OPENEDIT only verifies that the file is an EDIT file and creates the internal data structures necessary for subsequent IOEdit operations on the file. If *filenum* is a negative value or if the file does not exist, OPENEDIT opens the file by calling OPEN (after calling CREATE, if necessary) and then creates the internal IOEdit data structures.

On return, if OPENEDIT called OPEN, the returned value is the number returned by OPEN to identify the open file. If OPENEDIT did not call OPEN, the input value of *filenum* is returned unchanged.

flags input

INT:value

specifies the value that is passed to the *flags* parameter of the OPEN procedure if OPENEDIT opens the file. These conditions apply:

- If OPENEDIT opens the file and if the *flags* parameter is omitted, the value %002001 (read-only, shared access, nowait mode) is used.
- If OPENEDIT opens the file and if the *flags* parameter specifies write-only access, OPENEDIT opens the file with read-write access instead, because any write to an EDIT file requires reading a directory within the file to determine where to write the line.
- If the file is already open with write-only access at the time of the call to OPENEDIT, the procedure returns an error 2 (invalid operation), because it is unable to change the file's access mode.
- If the file does not exist and if the value of *flags* specifies (or defaults to) read-only access, OPENEDIT returns error 11 (file does not exist). If the file does not exist and if the *flags* value specifies read-write access, OPENEDIT creates the file and opens it.

For a detailed description of this parameter, see the *flags* parameter under OPEN Procedure (Superseded by FILE_OPEN_ Procedure).

sync-depth input

INT:value

specifies the sync depth value to be passed to the OPEN procedure if OPENEDIT opens the file. If this parameter is omitted, 0 is used.

For a detailed description of this parameter, see the *sync-or-receive-depth* parameter under OPEN Procedure (Superseded by FILE_OPEN_ Procedure).

write-thru input

INT:value

if present and not 0, specifies that each call to an IOEdit procedure that changes the content of the file should fully update the disk copy of the file. This means that every file access results in one or more physical I/O operations. Note that using this option can cause severe performance degradation.

Considerations

- The caller must set the *filenum* parameter to an appropriate value before each call to OPENEDIT, because its value might be changed upon return.
- If the file is already open at the time of the call to OPENEDIT, the *flags*, *sync-depth*, and *write-thru* parameters to OPENEDIT are ignored.
- OPENEDIT sets the file's current record number to -1 and resets the line number increment to 1 (that is, it resets the record number increment to 1000).
- If OPENEDIT calls the CREATE procedure, it sets the primary and secondary extent sizes to two pages each and sets the maximum number of extents to 900.
- If OPENEDIT opens a file that is already open by the same process, it writes to disk all the buffers for that file, including directory information. This assures that the file is in an up-to-date state at the completion of the open. For a general discussion of coordinating concurrent file access, see the *Guardian Programmer's Guide*.

Also Considerations on page 11-19.

Example

In this example, OPENEDIT calls OPEN for the file $MYVOL.TEST.AFILE. The default *flag* values are used (read-only, shared access, and nowait mode).

```
INT .EXT fname[0:11] := [ "$MYVOL  TEST    AFILE   " ];
INT .EXT fnumber := -1;
      .
      .
err := OPENEDIT ( fname, fnumber );
```

Related Programming Manual

For programming information about the IOEdit procedures, see the *Guardian Programmer's Guide*.

OPENEDIT_ Procedure

Summary

The OPENEDIT_ procedure allocates and initializes data blocks in the EDIT file segment (EFS) so that the specified file can be accessed later by the IOEdit procedures. It optionally creates and opens the specified file through the file system.

Syntax for C Programmers

```
#include <cextdecs(OPENEDIT_)>

short OPENEDIT_ ( const char *file-name
                 ,short length
                 ,short *filenum
                 ,[ short access ]
                 ,[ short exclusion ]
                 ,[ short nowait ]
                 ,[ short sync-depth ]
                 ,[ short write-thru ] );
```

Syntax for TAL Programmers

```
error := OPENEDIT_ ( file-name:length          ! i:i
                    ,filenum                   ! i,o
                    ,[ access ]                ! i
                    ,[ exclusion ]             ! i
                    ,[ nowait ]                ! i
                    ,[ sync-depth ]            ! i
                    ,[ write-thru ] );         ! i
```

Parameters

error returned value

INT

returns a file-system error number indicating the outcome of the operation. The possible values include:

-1 Page-count value is inconsistent.
-2 Page-table tags are out of order.
-3 Page-table tag is outside valid range.
-4 Page-table block number is outside of file.
-5 Page table has duplicate block numbers.

11 File does not exist; indicates that the file does not exist and that the *access* parameter indicates read-only access to the file.

14 Device does not exist; indicates that the device-name part of the file name designates a device that either does not exist or is not a disk device.

16 File has not been opened, wrong file type; indicates that the file is not an EDIT file (that is, the file type is not unstructured or the file code is not 101 or 102).

31 Unable to obtain buffer space; indicates that the file's directory does not fit into IOEdit's extended data segment and OPENEDIT_ is unable to enlarge the segment.

34 Unable to obtain memory space for control block; indicates that the number of IOEdit files already open is equal to the maximum number specified or assumed when INITIALIZEEDIT was called.

59 File is bad; indicates that the file exists and has the correct file type and file code for an EDIT file, but the data in the file has an incorrect format and OPENEDIT_ is unable to repair it.

The negative values listed above indicate that OPENEDIT_ has found a format error in the file that it can probably correct. At the time that one of these values is returned, the file has not yet been altered. If you immediately call CLOSEEDIT_ or CLOSEEDIT, the file is closed without change. If instead you call another IOEdit procedure, IOEdit tries to correct the format of the file. The format corrections are written to disk when CLOSEEDIT_ or CLOSEEDIT is finally called. If IOEdit fails to correct the format error, error 59 is returned for all subsequent IOEdit operations on the file.

file-name:*length* input:input

STRING .EXT:ref:*, INT:value

specifies the name of the file to be opened. If the file must be created, this name is assigned to it. The value of *file-name* must be exactly *length* bytes long and must be a valid file name or DEFINE name. If the name is partially qualified, it is resolved using the contents of the =_DEFAULTS DEFINE.

filenum input, output

INT:ref:1

on input, if the specified value is greater than or equal to 0 and if the file designated by *file-name* exists, indicates that the caller has already opened the file through the file system and that *filenum* is the number returned by the file system to identify the open file. In this case, OPENEDIT_ only verifies that the file is an EDIT file and creates the internal data structures necessary for subsequent IOEdit operations on the file. If *filenum* is a negative value or if the file does not exist, OPENEDIT_ opens the file by calling FILE_OPEN_ (after calling

FILE_CREATE_, if necessary), and then creates the internal IOEdit data structures.

On return, if OPENEDIT_ called FILE_OPEN_, the returned value is the number returned by FILE_OPEN_ to identify the open file. If OPENEDIT_ did not call FILE_OPEN_, the input value of *filenum* is returned unchanged.

access input

INT:value

specifies the value that is passed to the *access* parameter of the FILE_OPEN_ procedure if OPENEDIT_ opens the file. These conditions apply:

- If OPENEDIT_ opens the file and if the *access* parameter is omitted, 1 is used (indicating read-only access).
- If OPENEDIT_ opens the file and if the *access* parameter is equal to 2 (indicating write-only access), OPENEDIT_ opens the file with read-write access instead, because any write to an EDIT file requires reading a directory within the file to determine where to write the line.
- If the file is already open with write-only access at the time of the call to OPENEDIT_, the procedure returns an error 2 (invalid operation), because it is unable to change the file's access mode.
- If the file does not exist and if the *access* parameter is omitted or equal to 1 (indicating read-only access), OPENEDIT_ returns error 11 (file does not exist). If the file does not exist and if the *access* parameter is equal to 0 (indicating read-write access), OPENEDIT_ creates the file and opens it.

For a detailed description of this parameter, see the *access* parameter under FILE_OPEN_ Procedure.

exclusion input

INT:value

specifies the value that is passed to the *exclusion* parameter of the FILE_OPEN_ procedure if OPENEDIT_ opens the file. If this parameter is omitted, 0 is used (indicating shared access).

For a detailed description of this parameter, see the *exclusion* parameter under FILE_OPEN_ Procedure.

nowait input

INT:value

specifies the value that is passed to the *nowait-depth* parameter of the FILE_OPEN_ procedure if OPENEDIT_ opens the file. If this parameter is omitted, 1 is used (indicating that a maximum of one nowait I/O operation can be in progress at any time for this file).

For a detailed description of this parameter, see the *nowait-depth* parameter under FILE_OPEN_ Procedure.

For a description of how opening a file for nowait access affects an edit file, see Nowait Considerations on page 7-35.

sync-depth input

INT:value

specifies the sync depth value to be passed to the FILE_OPEN_ procedure if OPENEDIT_ opens the file. If this parameter is omitted, 0 is used.

For a detailed description of this parameter, see the *sync-or-receive-depth* parameter under FILE_OPEN_ Procedure.

write-thru input

INT:value

if present and not 0, specifies that each call to an IOEdit procedure that changes the content of the file should fully update the disk copy of the file. This means that every file access results in one or more physical I/O operations. Note that using this option can cause severe performance degradation.

Considerations

- The caller must set the *filenum* parameter to an appropriate value before each call to OPENEDIT_, because its value might be changed upon return.
- If the file is already open at the time of the call to OPENEDIT_, the *access*, *exclusion*, *nowait*, *sync-depth*, and *write-thru* parameters to OPENEDIT_ are ignored.
- OPENEDIT_ sets the file's current record number to -1 and resets the line number increment to 1 (that is, it resets the record number increment to 1000).
- If OPENEDIT_ calls the FILE_CREATE_ procedure, it sets the primary and secondary extent sizes to two pages each and sets the maximum number of extents to 900.
- If OPENEDIT_ opens a file that is already open by the same process, it writes to disk all the buffers for that file, including directory information. This assures that the file is in an up-to-date state at the completion of the open. For a general discussion of coordinating concurrent file access, see the *Guardian Programmer's Guide*.

Also see Considerations in the FILE_OPEN_ Procedure.

Example

In this example, OPENEDIT_ calls FILE_OPEN_ for the file $MYVOL.TEST.AFILE. These default values are used: read-only, shared access, nowait mode, sync depth of 0, and no unbuffered writes.

```
STRING .EXT fname[0:16] := [ "$MYVOL.TEST.AFILE" ];
INT  length := 17;
INT .EXT fnumber := -1;
      .
      .
err := OPENEDIT_ ( fname:length, fnumber );
```

Related Programming Manual

For programming information about the OPENEDIT_ procedure, see the *Guardian Programmer's Guide*.

OPENER_LOST_ Procedure

Summary

The OPENER_LOST_ procedure examines a system message and searches an open table to determine if an opener has been lost. An opener might be lost due to a processor or system failure that is reported in the system message.

An opener is a process that has opened the process that is calling OPENER_LOST_. An open table is a table that describes the opens (or openers) of the caller; it is created and maintained by the caller. See "Considerations," later in this subsection, for the description of an open table.

When a process receives a system message on $RECEIVE, it can call OPENER_LOST_ to determine if the message indicates that an opener has been lost. If OPENER_LOST_ finds that an opener has been lost, it deletes the table entry for that opener and returns the table index of the entry. The returned table index can be

supplied to OPENER_LOST_ in a subsequent call, along with the same system message, to search for additional lost openers.

Syntax for C Programmers

```
#include <cextdecs(OPENER_LOST_)>

short OPENER_LOST_ ( char *message
                    ,short length
                    ,short *table
                    ,short *index
                    ,short number-of-entries
                    ,short entry-size );
```

Syntax for TAL Programmers

```
status := OPENER_LOST_ ( message:length          ! i:i
                        ,table                   ! i
                        ,index                   ! i,o
                        ,number-of-entries       ! i
                        ,entry-size );           ! i
```

Parameters

status returned value

INT

indicates the result of the search. Valid values are:

0 Search completed; no lost openers

1 (reserved)

2 Parameter error

3 Bounds error

4 Backup opener lost

5 Primary opener lost; backup promoted to primary

6 Opener(s) lost; table entry now free

7 Message is not a relevant status-change message.

message:*length* input:input

STRING .EXT:ref:*, INT:value

is the status-change system message that was received. *message* must be exactly *length* bytes long. Relevant messages include:

-2	Local processor failure
-8	Network status change
-100	Remote processor failure
-110	Connection to remote system lost

If any other system message is supplied, an error is returned and the index is not advanced.

length is the length in bytes of the message.

table input

INT .EXT:ref:*

points to the beginning of the open table to be searched.

If the process handles of the primary and backup openers are not stored in the first 20 words of each table entry, *table* must point to the process handle of the first primary opener, not to the beginning of the table. See “Considerations.”

index input, output

INT .EXT:ref:*

on input, contains an index indicating at what point in the open table the search is to begin. On return, if a lost opener is discovered, *index* contains the index of that opener’s table entry; otherwise, its contents are undefined.

Ordinarily, you initialize *index* to -1 at the start of a search and do not alter its contents on subsequent calls (continuing the same search). See “Considerations.”

number-of-entries input

INT:value

contains the total number of entries in the open table.

entry-size input

INT:value

contains the size in words of each entry in the open table.

Considerations

- Determining if an opener has been lost

  The OPENER_LOST_ procedure reports that an opener has been lost if:

  - The connection to a remote system is lost and the process was running in that system.
  - A remote processor has failed and the process was running in that processor.
  - A local processor has failed and the process was running in that processor.

- The open table

  The open table is created and maintained by the caller. There should be an entry for each open of the caller, containing the process handles of the primary and backup openers plus any additional information that the caller wishes to store.

  The OPENER_LOST_ procedure makes these assumptions about the open table that is supplied to it:

  - Entries are of fixed length and are *entry-size* long.
  - The process handles of the primary and backup openers are stored back-to-back (the primary preceding the backup) in contiguous 10-word fields within each entry.
  - If the primary and backup process handles are not the first fields in the entry, *table* points to the first word of the first entry's primary-opener process handle rather than to the first word of the table entry.
  - An unused primary or backup field is marked with a null process handle (-1 in each word).

  When an opener is lost, OPENER_LOST_ makes any necessary updates to the primary and backup fields of the open table. If the backup opener is lost, the backup process handle is set to null. If the primary opener is lost, the backup process handle is moved to the primary opener field and the backup opener field is set to null (-1 in each word).

  These illustrates a possible layout for an opener table entry:

Section	Length
process handle of primary opener	10 words
process handle of backup opener	10 words
miscellaneous information	some fixed length

- Searching the open table

  To search the open table, call OPENER_LOST_ repeatedly until the returned *status* value is either 0 or a value that indicates an error condition. Before making the first call, initialize *index* to -1; do not alter its contents after that.

Example

```
index := -1;                                  ! initialize table
                                              !   index
```

```
DO                                          ! do until search
                                            !   is finished
   BEGIN                                    !   or an error
                                            !   occurs:
   status := OPENER_LOST_ ( message:length  ! search table
                           ,table^ptr,      !   for next
                           ,index,          !   affected
                           ,num^of^entries  !   entry
                           ,entry^size );
   do any necessary work
   END
UNTIL status = 0 OR status = 2 OR status = 3 OR status = 7;
```

OPENINFO Procedure (Superseded by FILE_GETOPENINFO_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The OPENINFO procedure obtains information about the opens of one disk file or of all the files on a disk device, or of certain nondisk devices. For these nondisk devices, only the *pricrtpid* and the *backcrtpid* parameters contain valid information. Each call returns information about one open; call OPENINFO successively to learn about all the opens.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
error := OPENINFO ( searchname                ! i
                  ,prevtag                    ! i,o
                  ,[ pricrtpid ]              ! o
                  ,[ backcrtpid ]             ! o
                  ,[ accessmode ]             ! o
                  ,[ exclusion ]              ! o
                  ,[ syncdepth ]              ! o
                  ,[ file-name ]              ! o
                  ,[ accessid ]               ! o
                  ,[ validmask ] );           ! o
```

Parameters

error returned value

INT

is the file-system error number indicating the outcome of the call. Error 1 (EOF) indicates there are no further opens. Error 2 (invalid operation) is returned for nondisk devices that cannot return any valid information.

searchname input

INT:ref:12

is the internal-format file name of the disk volume or disk file whose open information is being requested.

prevtag input, output

INT:ref:1

is a number identifying which open was last returned. Before making the first call, the user should set his *prevtag* variable to 0. On subsequent calls, the value should be that returned by the previous call.

pricrtpid output

INT:ref:4

is the process ID of the (primary) process that has the file open.

backcrtpid output

INT:ref:4

is the process ID of the backup process of a process-pair that has the file open, or is all zeros if there is no open by a backup.

accessmode output

INT:ref:1

is the access mode with which the file is open. The codes are:

0 read-write
1 read only
2 write only

exclusion output

INT:ref:1

is the exclusion mode with which the file is open. The codes are:

0 shared
1 exclusive
2 process exclusive (supported only for Optical Storage Facility)
3 protected

syncdepth output

INT:ref:1

is the sync depth specified when the file was opened.

file-name output

INT:ref:12

is the internal-format file name of the file which is open. This is of use when the *searchname* specified was not a file name.

accessid output

INT:ref:1

is the process access ID (user ID) of the opener at the time the open was done.

validmask output

INT:ref:1

returns a value indicating which of the information parameters had valid information returned. Each parameter has a corresponding bit in this value set to true if the parameter is valid for the device, as follows:

```
<0>   pricrtpid
<1>   backcrtpid
<2>   accessmode
<3>   exclusion
<4>   syncdepth
<5>   file-name
<6>   accessid
```

Considerations

- Order of returned information

  Opens are not returned in any defined order. In particular, when retrieving information about all opens on a disk volume, the opens for any one file are not grouped together in the sequence of calls.

- High-PIN considerations

  If a caller uses OPENINFO to obtain the process ID of a primary or backup process that has a high PIN, the returned *validmask* bit for that process ID is 0 and the returned process ID value is all zeros. Thus, if the primary process has a high PIN, *validmask*.<0> = 0 and *pricrtpid* is zero-filled; if the backup process has a high PIN, *validmask*.<1> = 0 and *backcrtpid* is zero-filled.

- Support for HP NonStop Storage Management Foundation (SMF) objects

  The OPENINFO procedure supports single SMF logical files but does not support entire SMF virtual volumes. If the name of a SMF logical file is supplied to this procedure, the system queries the disk process of the appropriate physical volume to obtain information about current openers. If the name of a SMF virtual volume is supplied, but not a full logical file name, an error is returned.

  If you call the OPENINFO procedure and supply the name of a physical volume that has an open that was made on a SMF logical file name, information about the open is returned, but the returned file name is that of the physical file supporting the logical file.

OSS Considerations

It is often necessary to run OSS processes at high PINs. See “Considerations,” earlier, for more information.

Example

```
DEVICE^NAME^PADDED ':=' ["$DEVICE ",8*["   "]];
NUM := 0;

DO -- Get OPENINFO for the device
  BEGIN
  ERROR := OPENINFO( DEVICE^NAME^PADDED,
                     NUM,
                     PCRTPID,
                     BCRTPID,
                     ACCESSMODE,
                     EXCLUSION,
                     SYNC,
                     FILENAME,
                     ACCESSID );
  IF ERROR = 0  !SUCCESS! THEN
    BEGIN
    --    Process (filter/sort) OPEN-record
    END;
  END -- Get OPENINFO for the device
UNTIL ERROR <> 0  !SUCCESS!;

IF ERROR = 2  !INVALID OPERATION! THEN
  BEGIN
  --    Device doesn't support OPENINFO
  END
ELSE IF ERROR <> 1  !END OF FILE! THEN
  BEGIN
  --    File-system error/resource problem
  END;
```

OSS_PID_NULL_ Procedure

Summary

The OSS_PID_NULL_ procedure returns a null OSS process ID.

Syntax for C Programmers

```
#include <cextdecs(OSS_PID_NULL_)>

__int32_t OSS_PID_NULL_ ( void );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
oss-pid := OSS_PID_NULL_;
```

Parameters

oss-pid returned value

INT (32)

is a null OSS process ID.

OSS Considerations

A null OSS process ID can be passed to a procedure such as PROCESS_GETINFOLIST_ to indicate that the OSS process ID parameter is not present (an alternative to omitting the parameter). The value of the null OSS process ID can change from one RVU to another.

12 Guardian Procedure Calls (P)

This section contains detailed reference information for all user-accessible Guardian procedure calls beginning with the letter P. Table 12-1 lists all the procedures in this section.

Table 12-1. Procedures Beginning With the Letter P (page 1 of 2)

Table 12-1. Procedures Beginning With the Letter P (page 2 of 2)

PROCESSHANDLE_GETMINE_ Procedure
PROCESSHANDLE_NULLIT_ Procedure
PROCESSHANDLE_TO_CRTPID_ Procedure
PROCESSHANDLE_TO_FILENAME_ Procedure
PROCESSHANDLE_TO_STRING_ Procedure
PROCESSINFO Procedure (Superseded by PROCESS_GETINFOLIST_ Procedure)
PROCESSNAME_CREATE_ Procedure
PROCESSOR_GETINFOLIST_ Procedure
PROCESSOR_GETNAME_ Procedure
PROCESSORSTATUS Procedure
PROCESSORTYPE Procedure
PROCESSSTRING_SCAN_ Procedure
PROCESSTIME Procedure (Superseded by PROCESS_GETINFOLIST_ Procedure)
PROGRAMFILENAME Procedure (Superseded by PROCESS_GETINFOLIST_ Procedure)
PURGE Procedure (Superseded by FILE_PURGE_ Procedure)
PUTPOOL Procedure (Superseded by POOL_* Procedures)

PACKEDIT Procedure

Summary

The PACKEDIT procedure converts a line image from unpacked format into EDIT packed line format. The input value is a text string that can include sequences of blank characters; the returned value is the same text in packed format with blank compression codes.

PACKEDIT is an IOEdit procedure and is intended for use with files that have been opened by OPENEDIT or OPENEDIT_.

Syntax for C Programmers

```
#include <cextdecs(PACKEDIT)>

void PACKEDIT ( char *unpacked-line
                ,short unpacked-length
                ,char *packed-line
                ,short packed-limit
                ,short *packed-length
                ,[ short full-length ] );
```

Syntax for TAL Programmers

```
CALL PACKEDIT ( unpacked-line              ! i
                ,unpacked-length           ! i
                ,packed-line               ! o
                ,packed-limit              ! i
                ,packed-length             ! o
                ,[ full-length ] );        ! i
```

Parameters

unpacked-line input

STRING .EXT:ref:*

is a string array that contains the line in unpacked format that is to be converted. The length of *unpacked-line* is specified by the *unpacked-length* parameter.

unpacked-length input

INT:value

specifies the length in bytes of *unpacked-line*.

packed-line output

STRING .EXT:ref:*

is a string array that contains the line in packed format that is the outcome of the conversion. The length of the packed line is returned in the *packed-length* parameter.

packed-limit input

INT:value

specifies the length in bytes of the string variable *packed-line*.

packed-length output

INT .EXT:ref:1

returns the actual length in bytes of the value returned in *packed-line*. If *packed-line* is not large enough to contain the value that is the output of the conversion, *packed-length* returns a value of -1.

full-length input

INT:value

if present and not equal to 0, specifies that all trailing space characters (if any) in the line being processed should be retained in the output line image. Otherwise, trailing space characters are discarded.

Considerations

- If a line contains few sequences of blank characters, it might require more bytes in packed format than in unpacked format. To provide for this, you should specify a value for *packed-limit* that is at least 8% greater than *unpacked-length*.

Related Programming Manual

For programming information about the PACKEDIT procedure, and for a description of the EDIT packed line format, see the *Guardian Programmer's Guide*.

PATHNAME_TO_FILENAME_ Procedure

Summary

The PATHNAME_TO_FILENAME_ procedure converts an OSS pathname to a Guardian file name. For a description of the OSS pathname syntax, see Appendix D, File Names and Process Identifiers.

Syntax for C Programmers

```
#include <cextdecs(PATHNAME_TO_FILENAME_)>

short PATHNAME_TO_FILENAME_ ( const char *path
                             ,char *filename
                             ,short maxlen
                             ,short *length
                             ,[ short *infoflags ] );
```

Syntax for TAL Programmers

```
error := PATHNAME_TO_FILENAME_ ( pathname              ! i
                                ,filename:maxlen       ! o:i
                                ,length                ! o
                                ,[ info-flags ] );     ! o
```

Parameters

error returned value

INT

returns a file-system error number indicating the outcome of the operation:

0	No error.
563	The buffer pointed to by *filename* is too small.
4002	No such *pathname* exists. The corresponding OSS errno value is ENOENT.

4006 A prefix within *pathname* refers to an OSS fileset other than the root fileset that is not mounted. The corresponding OSS errno value is ENXIO.

4013 Search permission is denied on a component of the pathname prefix. The corresponding OSS errno value is EACCESS.

4014 A specified parameter has an invalid address. The corresponding OSS errno value is EFAULT.

4020 A prefix within *pathname* refers to a file other than a directory. The corresponding OSS errno value is ENOTDIR.

4022 *pathname* is invalid. The corresponding OSS errno value is EINVAL.

4131 The pathname, a component of the pathname, or a symbolic link in the pathname is longer than PATH_MAX characters. (PATH_MAX is a symbolic constant that is defined in the OSS limitsh header file.) For pathname syntax, see Appendix D, File Names and Process Identifiers. The corresponding OSS errno value is ENAMETOOLONG.

4200 The pathname or a component of the pathname has too many symbolic links to resolve the specified pathname. The corresponding OSS errno value is ELOOP.

4202 The root fileset is not mounted. The corresponding OSS errno value is ENOROOT.

4203 OSS is not installed or is not initialized. The corresponding OSS errno value is EOSSNOTRUNNING.

pathname input

STRING .EXT:ref:*

is the null-terminated OSS pathname to be converted into its corresponding Guardian file name.

filename:*maxlen* output:input

STRING .EXT:ref:*, INT:value

returns the Guardian file name that corresponds to *pathname*. The file name is not null-terminated; its length is returned in *length*.

maxlen specifies the maximum length in bytes of the name that can be returned in *filename*. If *maxlen* is not large enough, *error* returns 563 (buffer too small) and *length* returns the actual length of the name.

filename contains a null string if *pathname* does not correspond to a Guardian file name. In this case, the value returned in *error* is 0.

length output

INT .EXT:ref:1

returns the length in bytes of the fully qualified Guardian file name returned in *filename*. If *error* returns 563 (buffer too small) due to *filename* being to small to contain the name, *length* returns the actual length of the name.

info-flags output

INT .EXT:ref:1

contains additional information about the file. *info-flags* is returned as a bit mask defined as:

<0:14>	Reserved	
<15>	= 1	The specified file is a Guardian file.
	= 0	The specified file is an OSS file.

OSS Considerations

- If the file identified by *pathname* is in the Guardian name space (/G), then the file name is syntactically changed to the Guardian format without checking whether the file exists. The local pathname of a permanent Guardian disk file has the form /G/*volume*/*subvol*/*file-id* which corresponds to the Guardian name $*volume*.*subvol*.*file-id*. Similarly, the local pathname for a temporary Guardian disk file has the form /G/volname/#number which corresponds to the Guardian name $volume.#number. The conversion takes place as follows:
  - The initial “/G/” is removed.
  - The remaining slash separators (/) are replaced by periods.
  - If the current directory symbol (.) is part of the pathname it is safely ignored.
  - If the parent directory symbol (..) is part of the pathname, the first element to the left is deleted.
  - A leading dollar sign ($) is added for part of the Guardian volume name.
  - Any period (.), hyphen (-), or underscore (_) characters within pathname elements are deleted.
  - Name elements are truncated to eight characters after the “.”, “-”, and “_” characters are deleted.
- No timestamps are updated as a result of this procedure.
- Two additional file numbers might be allocated: one for the OSS root directory and one for the OSS current working directory. These files are not necessarily the next available file numbers and they cannot be closed by calling FILE_CLOSE_.
- A current OSS working directory is established from the value of the VOLUME attribute of the =_DEFAULTS DEFINE.

- The resident memory used by the calling process increases by a small amount.

Example in C

```
ret = PATHNAME_TO_FILENAME_(
      argv[1],                  /* OSS Pathname */
      filename,                 /* Guardian file name buffer */
      64,                       /* size of file name buffer */
      &filelen,                 /* length of file name */
      &status );                /* if = 1, Guardian file (/G)
                                   if = 0, OSS file */
```

Related Programming Manual

For programming information about the PATHNAME_TO_FILENAME_ procedure, see the *Open System Services Programmer's Guide*.

POOL_CHECK_ Procedure

Summary

The POOL_CHECK_ procedure checks the internal pool data structures and returns error information.

Syntax for C Programmers

```
#include <cextdecs(POOL_CHECK_)>

short POOL_CHECK_ ( short *pool
                  ,[ __int32_t *corruption-address ]
                  ,[ __int32_t *block ]
                  ,[ __int32_t *block-size  ]
                  ,[ short *tag-size  ] );
```

- CEXTDECS (through the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
error := POOL_CHECK_ ( pool                              ! i
                     ,[ corruption-address ]             ! o
                     ,[ block ]                          ! o
                     ,[ block-size ]                     ! o
                     ,[ tag-size ] );                    ! o
```

Parameters

error returned value

INT

indicates the outcome of the call:

0 No error.

2 Required parameter missing. *pool* must be specified.

3 Bounds error. A parameter on the parameter list has a bounds error.

9 Corrupt pool header.

11 Corrupt allocated blocks. Data is probably written beyond the allocated block.

12 Corrupt free list blocks. Data is probably written into a returned block.

pool input

INT .EXT:ref:*

is the address of the pool as specified in the call to the POOL_DEFINE_ procedure.

corruption-address output

EXTADDR .EXT:ref:1

is defined for these values of *error*:

error	***corruption-address***
11	Address of the allocated block where the corruption is detected.
12	Address of the free block where the corruption is detected.

block output

EXTADDR .EXT:ref:1

is defined for these values of *error*:

error	***block***
11	Address of the valid allocated block that precedes the address where the corruption is detected. If the corruption is detected in the first block, then *block* is -4D.
12	Address of the valid free block that precedes the address where the corruption is detected. If the corruption is detected in the first block, then *block* is -4D.

block-size output

INT .EXT:ref:1

is defined for these values of *error*:

error	***block-size***
11	Block size of the last valid allocated block that precedes the address where the corruption is detected. If the corruption is detected in the first block, then *block* is -4D and *block-size* is undefined.
12	Block size of the last valid free block that precedes the address where the corruption is detected. If the corruption is detected in the first block, then *block* is -4D and *block-size* is undefined.

tag-size output

INT .EXT:ref:1

is the size in bytes of a boundary tag that defines the beginning or end of a block.

Considerations

See “Considerations” in POOL_DEFINE_ Procedure.

Example

```
error := POOL_CHECK_( pool, corruption^addr, pblock );
```

Related Programming Manual

For programming information about the POOL_CHECK_ memory-management procedure, see the *Guardian Programmer’s Guide*.

POOL_DEFINE_ Procedure

Summary

The POOL_DEFINE_ procedure designates a portion of a user's stack or an extended data segment for use as a pool.

Syntax for C Programmers

```
#include <cextdecs(POOL_DEFINE_)>

short POOL_DEFINE_ ( short *pool
                    ,__int32_t pool-size
                    ,[ short alignment ]
                    ,[ short priv-only ] );
```

- CEXTDECS (through the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
error := POOL_DEFINE_ ( pool                          ! i
                       ,pool-size                     ! i
                       ,[ alignment ]                 ! i
                       ,[ priv-only ] );              ! i
```

Parameters

error returned value

INT

indicates the outcome of the call:

0 No error.

2 Required parameter missing. Pool and pool-size must be specified or a non-privileged caller specified a non-zero value for priv-only.

3 Bounds error. *pool* is in a read-only segment, or *pool-size* is larger than the space available.

4 Invalid size. *pool-size* is too small to allocate the minimum size pool including the pool header.

5 Alignment error on *pool*. *pool* is not in alignment with the selected *alignment*.

6 Invalid alignment. *alignment* is not 0, 4, 8, or 16.

pool input

INT .EXT:ref:*

specifies the address of the first word of the memory space to be used as the pool, including the pool header. The address must be aligned according to *alignment;* the default alignment is 8 bytes.

pool-size input

INT(32):value

specifies the size of the pool, including the pool header, in bytes. The maximum size is limited only by the amount of space available to the application. The address of the end of the pool is always equal to the address specified for *pool* plus the value of *pool-size*. Pool space overhead and adjustments for alignment do not cause the pool to extend past this boundary.

alignment input

INT:value

specifies the alignment of blocks allocated from the pool.

0 8 byte alignment
4 4 byte alignment
8 8 byte alignment
16 16 byte alignment (the default alignment)

On TNS processors, an 8-byte alignment is recommended.

On native processors, a smaller *alignment* generates the most compact pool with the least cache alignment, and a larger *alignment* generates the least compact pool with most cache alignment.

priv-only input

This parameter can only be used by a privileged caller.

Considerations

Note. There are additional considerations for privileged callers.

- Internal variable-length pool header

  The POOL_DEFINE_ procedure creates an internal variable-length pool header at the beginning of the pool. The length of the header can change between RVUs or processor types. Information in the header is retrieved by calling the POOL_GETINFO_ procedure; the header should not be accessed directly, since it is subject to change.

- Stack addresses converted to extended addresses

  If the pool is in the user data stack, the TAL compiler automatically converts data stack addresses to extended addresses.

- Read-only segments

  If you specify a pool in an extended data segment that is allocated as a read-only segment, the POOL_DEFINE_ procedure returns error 3 (bounds error).

- Dynamic memory allocation

  Several Guardian procedures support the creation of memory pools and the dynamic allocation of variable-sized blocks from a pool. The calling program provides the memory area to be used as the pool and then calls the POOL_DEFINE_ procedure to initialize the pool. The pool can reside in the user data stack or in an extended data segment. The pool procedures accept and return extended addresses that apply to both the stack and extended memory.

  Once the pool is defined, the process can reserve blocks of various sizes from the pool by calling the POOL_GETSPACE_ procedure and can release blocks by calling the POOL_PUTSPACE_ procedure. The program must release one entire block in a POOL_PUTSPACE_ call; it cannot release part of a block or multiple blocks in one POOL_PUTSPACE_ call. If the pool is too small or is larger than necessary, the process can resize the pool by calling the POOL_RESIZE_ procedure. For detecting potential problems with the pool, the POOL_GETINFO_ procedure returns information about a pool and the POOL_CHECK_ procedure checks the internal data structures of a pool.

  Be careful to use only the currently reserved blocks of the pool. Using blocks that are not reserved causes pool corruption. If multiple pools are defined, make sure to return reserved blocks to the correct pool. For debugging purposes, call POOL_GETINFO_ for information on the pool header and call POOL_CHECK_ to check the pool for consistency.

- Pool management methods

  This information is supplied for use in evaluating the appropriateness of using the Guardian pool routines in user application programs and in determining the proper size of a pool.

Each block allocated by the POOL_GETSPACE_ procedure has a tag at the beginning of the block and a tag at the end of the block. A block boundary tag serves three purposes:

- It contains the size of each block so that the program does not need to specify the length of a block when releasing it.
- It serves as a check to ensure that the program does not erroneously use more memory than the block contains (although it does not stop the program from overwriting).
- It provides for efficient coalescing of adjacent free blocks.

- POOL can only be defined on a single segment. It cannot be defined from segment space of two consecutive logical segments.

  The pool space overhead on each block can be substantial if very small blocks are allocated (in current RVUs, the minimum block size is 32 bytes).

  Although pools can also be used to manage the allocation of a collection of equal-sized blocks, these procedures are not recommended for that purpose because they can consume more processor time and pool memory than user-written routines designed for that specific task.

Example

```
error := POOL_DEFINE_ ( pool, 2048D );
```

Related Programming Manual

For programming information about the POOL_DEFINE_ memory-management procedure, see the *Guardian Programmer's Guide*.

POOL_GETINFO_ Procedure

Summary

The POOL_GETINFO_ procedure returns information about the specified pool.

Syntax for C Programmers

```
#include <cextdecs(POOL_GETINFO_)>

short POOL_GETINFO_  ( short *pool
                      ,[ short *error-detail ]
                      ,[ __int32_t *avail-pool-size ]
                      ,[ __int32_t *curalloc ]
                      ,[ __int32_t *maxalloc ]
                      ,[ __int32_t *fail-block-size ]
                      ,[ short *curfrag ]
                      ,[ short *maxfrag ]
                      ,[ short *alignment ]
                      ,[ short *tag-size] );
```

- CEXTDECS (through the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
error := POOL_GETINFO_ ( pool                          ! i
                        ,[ error-detail ]              ! o
                        ,[ avail-pool-size ]           ! o
                        ,[ curalloc ]                  ! o
                        ,[ maxalloc ]                  ! o
                        ,[ fail-block-size ]           ! o
                        ,[ curfrag ]                   ! o
                        ,[ maxfrag ]                   ! o
                        ,[ alignment ]                 ! o
                        ,[ tag-size] );                ! o
```

Parameters

error returned value

INT

indicates the outcome of the call:

0 No error

2 Required parameter missing. *error-detail* contains the number of the first parameter found to be in error, where 1 designates the first parameter on the left.

3 Bounds error. *error-detail* contains the number of the first parameter found to be in error, where 1 designates the first parameter on the left.

9 Corrupt pool header.

pool input

INT .EXT:ref:*

is the address of the pool as specified in the call to the POOL_DEFINE_ procedure.

error-detail output

INT .EXT:ref:1

for some returned errors, contains additional information. See *error*.

avail-pool-size output

INT(32) .EXT:ref:1

is the available space in bytes in the pool, not including the pool header.

Note that the value of the *pool-size* parameter specified in the POOL_DEFINE_ procedure is larger than this value because it includes the pool header.

curalloc output

INT(32) .EXT:ref:1

is the current amount of space allocated from the pool in bytes, not including the pool header.

maxalloc output

INT(32) .EXT:ref:1

is the maximum amount of space ever allocated from the pool, in bytes, since it was originally allocated with the POOL_DEFINE_ procedure. *maxalloc* does not include the pool header.

fail-block-size output

INT(32) .EXT:ref:1

is the size in bytes of the block that the POOL_GETSPACE_ procedure failed to allocate when it returned an *error* value of 10 (unable to allocate space). Note that the value of the *block-size* parameter specified in the POOL_GETSPACE_ procedure is smaller than *fail-block-size* because it is not rounded up. For a description of the differences between the requested block size and the allocated block size, see Considerations on page 12-13.

curfrag output

INT .EXT:ref:1

is the number of free fragments in the pool.

maxfrag output

INT .EXT:ref:1

is the maximum number of free fragments that the pool has ever had since it was originally allocated with the POOL_DEFINE_ procedure.

alignment output

INT .EXT:ref:1

is the pool alignment selected when the pool was originally allocated with the POOL_DEFINE_ procedure.

tag-size output

INT .EXT:ref:1

is the size in bytes of one of the tags used to mark the free or allocated blocks.

Considerations

Note. There are additional considerations for privileged callers.

See Considerations on page 12-13.

Example

```
error :=
   POOL_GETINFO_ ( pool, error^detail, avail^pool^size );
   ! determine the available pool size
```

Related Programming Manual

For programming information about the POOL_GETINFO_ memory-management procedure, see the *Guardian Programmer's Guide*.

POOL_GETSPACE_ Procedure

Summary

The POOL_GETSPACE_ procedure obtains a block of memory from a buffer pool.

Syntax for C Programmers

```
#include <cextdecs(POOL_GETSPACE_)>

__int32_t POOL_GETSPACE_ ( short *pool
                          ,__int32_t block-size
                           ,short *error );
```

- CEXTDECS (through the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
block := POOL_GETSPACE_ ( pool                          ! i
                         ,block-size                    ! i
                         ,[ error ] );                  ! o
```

Parameters

block returned value

INT(32)

returns the extended address of the first byte in the memory block obtained if the operation is successful, and returns %37777000000D if an error occurs.

pool input

INT .EXT:ref:*

is the address of the pool as specified in the call to the POOL_DEFINE_ procedure. When POOL_GETSPACE_ is called, the pool header is updated.

block-size input

INT(32):value

is the size in bytes of the memory to be obtained from the pool. This value can range from 1 byte through the available space in the pool. The block size of the allocated block can be rounded up to retain the alignment of the pool.

error output

INT .EXT:ref:1

indicates the outcome of the call:

0 No error.

2 Required parameter missing.

4 Invalid size. *block-size* is not within the valid range.

9 Corrupt pool header.

10 Unable to allocate space.

Considerations

Note. There are additional considerations for privileged callers.

POOL_GETSPACE_ and POOL_PUTSPACE_ do not check pool data structures on each call. A process that destroys data structures or uses an incorrect address for a parameter can fail on a call to POOL_GETSPACE_ or POOL_PUTSPACE_: a TNS process can get an instruction failure trap (trap 1) or invalid address trap (trap 0); a native process can receive a SIGILL or SIGSEGV signal.

Example

```
@pblock := POOL_GETSPACE_( pool, $UDBL( $LEN( pblock ) ) );
    ! get a pool block of PBLOCK size.
```

Related Programming Manual

For programming information about the POOL_GETSPACE_ memory-management procedure, see the *Guardian Programmer's Guide*.

POOL_GETSPACE_PAGE_ Procedure (H-Series RVUs Only)

Summary

The POOL_GETSPACE_PAGE_ procedure obtains a block of memory from a buffer pool. The memory is aligned on a page boundary and the space allocated is a multiple of a page size.

Syntax for C Programmers

```
#include <cextdecs(POOL_GETSPACE_PAGE_)>

__int32_t POOL_GETSPACE_PAGE_ ( short *pool
                              ,__int32_t block-size
                              ,short *error );
```

- CEXTDECS (through the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
block := POOL_GETSPACE_PAGE_ ( pool                    ! i
                             ,block-size               ! i
                             ,[ error ] );             ! o
```

Parameters

block returned value

INT(32)

returns the extended address of the first byte in the memory block obtained if the operation is successful, and returns %37777000000D if an error occurs.

pool input

INT .EXT:ref:*

is the address of the pool as specified in the call to the POOL_DEFINE_ procedure. When POOL_GETSPACE_PAGE_ is called, the pool header is updated.

error output

INT .EXT:ref:1

indicates the outcome of the call:

0 No error.

2 Required parameter missing.

4 Invalid size. *size* is not within the valid range.

9 Corrupt pool header.

10 Unable to allocate space.

POOL_PUTSPACE_ Procedure

Summary

The POOL_PUTSPACE_ procedure returns a block of memory to a buffer pool.

Syntax for C Programmers

```
#include <cextdecs(POOL_PUTSPACE_)>

short POOL_PUTSPACE_ ( short *pool
                      ,short *block );
```

Syntax for TAL Programmers

```
error := POOL_PUTSPACE_ ( pool                          ! i
                         ,block );                      ! i
```

Parameters

error returned value

INT

indicates the outcome of the call:

0 No error.

2 Required parameter missing.

3 Bounds error: *block* is not within the pool boundaries.

9 Corrupt pool header.

11 Corrupt allocated block: Data is probably written beyond the allocated block or the block has already been returned.

pool input

INT .EXT:ref:*

is the address of the pool as specified in the call to the POOL_DEFINE_ procedure. When POOL_PUTSPACE_ is called, the pool header is updated.

block input

INT .EXT:ref:*

is the address of the block to be returned to the pool.

Considerations

POOL_GETSPACE_ and POOL_PUTSPACE_ do not check pool data structures on each call. A process that destroys data structures can fail on a call to POOL_GETSPACE_ or POOL_PUTSPACE_: a TNS Guardian process can get a bounds violation trap (trap 0); an OSS or native process can receive a SIGSEGV signal.

Example

```
error := POOL_PUTSPACE_ ( pool, pblock );
    ! put a block obtained from POOL_GETSPACE_ back into
    ! the pool obtained from POOL_DEFINE_.
```

Related Programming Manual

For programming information about the POOL_PUTSPACE_ memory-management procedure, see the *Guardian Programmer's Guide*.

POOL_RESIZE_ Procedure

Summary

The POOL_RESIZE_ procedure changes the size of a pool that was initialized by the POOL_DEFINE_ procedure.

Syntax for C Programmers

```
#include <cextdecs(POOL_RESIZE_)>

short POOL_RESIZE_ ( short *pool
                    ,__int32_t new-pool-size );
```

- CEXTDECS (through the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
error  :=  POOL_RESIZE_ ( pool                          ! i
                         ,new-pool-size );              ! i
```

Parameters

error returned value

INT

indicates the outcome of the call:

0 No error.

2 Required parameter missing.

3 Bounds error. *pool* is in a read-only segment, or *new-pool-size* is larger than the available space.

4 Invalid size. *new-pool-size* is too small to allocate the minimum size pool, including the pool header.

9 Corrupt pool header.

11 Corrupt allocated blocks. Data is probably written beyond the allocated block.

12 Corrupt free list blocks. Data is probably written into a returned block.

13 Unable to shrink pool.

pool input

INT .EXT:ref:*

is the address of the pool as specified in the call to the POOL_DEFINE_ procedure. When POOL_RESIZE_ is called, the pool header is updated.

new-pool-size input

INT(32):value

specifies the new size of the pool, including the pool header, in bytes. The maximum size is limited only by the amount of space available to the application. The address of the end of the pool is always equal to the address specified for the *pool* parameter plus the value of the *new-pool-size* parameter. Pool space overhead and adjustments for alignment do not cause the pool to extend past this boundary.

Considerations

See Considerations on page 12-13.

Example

```
error := POOL_RESIZE_ ( pool, 4096D );
```

Related Programming Manual

For programming information about the POOL_RESIZE_ memory-management procedure, see the *Guardian Programmer's Guide*.

POSITION Procedure (Superseded by FILE_SETPOSITION_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The POSITION procedure positions by primary key within relative and entry-sequenced files. For unstructured files, the POSITION procedure specifies a new current position.

For relative and unstructured files, POSITION sets the current position, access path, and positioning mode for the specified file. The current position, access path, and positioning mode define a subset of the file for subsequent access.

The POSITION procedure is not used with key-sequenced files; KEYPOSITION is used instead.

The caller is not suspended because of a call to POSITION.

A call to the POSITION procedure is rejected with an error indication if there are incomplete nowait operations pending on the specified file.

Syntax for C Programmers

```
#include <cextdecs(POSITION)>

_cc_status POSITION ( short filenum
                     ,__int32_t record-specifier );
```

- CEXTDECS (through the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.
- The function value returned by POSITION, which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL POSITION ( filenum                      ! i
               ,record-specifier );          ! i
```

Parameters

filenum input

INT:value

is the number of an open file that identifies the file where the positioning is to take place.

record-specifier input

INT(32):value

is the 4-byte value that specifies the new setting for the current-record and next-record pointers.

Relative Files	*record-specifier* is a 4-byte *record-num*..
	-2D specifies that the next write should occur at an unused record position.
	-1D specifies that subsequent writes should be appended to the end-of-file location.
Unstructured Files	*record-specifier* is a 4-byte *relative-byte-addr*.
	-1D or -2D specifies that subsequent writes should be appended to the EOF location.

(For relative and unstructured files, the -1D and -2D remain in effect until a new *record-specifier* is supplied.)

Entry-Sequenced Files

The *record-specifier* is a 4-byte record-addr (the primary key), whose format depends upon the file's block size as follows:

Block size	Number of bits for block number	Number of bits for the relative record number within that block
4096	20	12
2048	21	11
1024	22	10
512	23	9

In all cases, the block number occupies the leftmost bits, and the record number occupies the rightmost bits.

For information about *record-addr*, see the *Enscribe Programmer's Guide*.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILE_GETINFO_ or FILEINFO).

= (CCE) indicates that the POSITION was successful.

> (CCG) indicates no operation; *filenum* does not designate a disk file.

Considerations

- POSITION does not cause the disk heads to be repositioned (at least until a subsequent data transfer is initiated).
- POSITION cannot be used with Enscribe format 2 and OSS files if the file was opened with the "Use 64 bit keys" choice to FILE_OPEN_ (which is necessary to access a file with a file size of over 2 GB). If an attempt is made to use the POSITION procedure with such files, error 581 is returned. For information on how to perform the equivalent task with Enscribe format 2 files and OSS files greater than approximately 2 gigabytes, see the FILE_SETPOSITION_ Procedure.
- Unstructured files
  - File pointers after POSITION

    After a successful call to POSITION for an unstructured file, the file pointers are:

    ```
    current^record^pointer := record-specifier;
    next^record^pointer := record-specifier;
    ```

  - Value of *record-specifier* for unstructured files

Unless the unstructured file is created with the odd unstructured attribute (also known as ODDUNSTR) set, the RBA passed in *record-specifier* must be an even number. If the odd unstructured attribute is set when the file is created, the RBA passed in *record-specifier* can be either an odd or even value. (You set the odd unstructured attribute with the FILE_CREATE_, FILE_CREATELIST_, or CREATE procedure, or with the File Utility Program (FUP) SET and CREATE commands.)

For even unstructured files (that is, files created with the odd unstructured attribute not set), the *record-specifier* parameter must be an even byte address, or the operation fails with file-system error 2.

- Relative and entry-sequenced files
  - Writing to entry-sequenced files

    Inserts to entry-sequenced files always occur at the end of the file.
  - Current-state indicators for structured files

    After a successful POSITION to a relative or entry-sequenced file, the current-state indicators are:

    Current position is that of the record indicated by the *record-specifier*.

    Positioning mode is approximate.

    Comparison length is 4.

    Current primary-key value is set to the value of the *record-specifier*.

Related Programming Manuals

For programming information about the POSITION file-system procedure, see the *Enscribe Programmer's Guide* and the *Guardian Programmer's Guide*.

POSITIONEDIT Procedure

Summary

The POSITIONEDIT procedure sets the next record number to a specified value for a specified file.

POSITIONEDIT is an IOEdit procedure and can only be used with files that have been opened by OPENEDIT or OPENEDIT_.

Syntax for C Programmers

```
#include <cextdecs(POSITIONEDIT)>

short POSITIONEDIT ( short filenum
                    ,__int32_t record-number );
```

- CEXTDECS (through the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
error := POSITIONEDIT ( filenum                    ! i
                       ,record-number );           ! i
```

Parameters

error returned value

INT

returns a file-system error number indicating the outcome of the operation.

filenum input

INT:value

specifies the file number of the open file for which the next record number is to be set.

record-number input

INT(32):value

specifies the value to which the file's next record number is to be set. This value is 1000 times the EDIT line number of the intended record. You can specify -1 to indicate positioning to the beginning of the file; you can specify -2 to indicate positioning to the end of file.

Example

In this example, POSITIONEDIT sets the next record number to indicate line 500 in the specified file.

```
INT(32) next-record-number := 500000D;
      .
      .
error := POSITIONEDIT ( filenumber, next-record-number );
```

Related Programming Manual

For programming information about the POSITIONEDIT procedure, see the *Guardian Programmer's Guide*.

PRIORITY Procedure (Superseded by PROCESS_SETINFO_ Procedure or PROCESS_GETINFOLIST_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The PRIORITY procedure enables a process to examine or change its initial priority. The current priority is updated to the initial priority value when the process waits for an external event to occur.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
old-priority := PRIORITY ( [ new-priority ]          ! i
                         ,[ init-priority ] );       ! o
```

Parameters

old-priority returned value

INT

returns a value that is either:

- The current priority of the process if *new-priority* is not specified

- The previous value of the current priority when *new-priority* is specified
- A 0, indicating that the specified *new-priority* value is out of range and that the priority was not changed

new-priority input

INT:value

specifies a new execution priority value in the range {1:199} for this process. If omitted, the initial priority remains unchanged.

init-priority output

INT:ref:1

returns the initial run priority of the process when it was started.

Considerations

- A caller of PRIORITY executing in privileged mode can set its priority to a value greater than 199. However, if such a process has a priority greater than that of the memory-manager process and gets a memory page fault, the call to PRIORITY fails: a Guardian TNS process gets a “no memory available” trap (trap 12); an OSS or native process receives a SIGNOMEM signal.
- The current priority rather than the initial priority is returned. Due to the sliding priority feature on NonStop servers, the current priority may be lower than the initial priority if the process is processor-bound (that is, the process does not perform any I/O requests while running).

Example

```
LAST^PRI := PRIORITY ( 100 ); ! changes the current
                              ! priority to 100.
```

PROCESS_ACTIVATE_ Procedure

Summary

The PROCESS_ACTIVATE_ procedure returns a suspended process or process pair to the ready state. A process is suspended by calling the PROCESS_SUSPEND_ or SUSPENDPROCESS procedure, or by entering a TACL SUSPEND command.

Syntax for C Programmers

```
#include <cextdecs(PROCESS_ACTIVATE_)>

short PROCESS_ACTIVATE_ ( short *processhandle
                        ,[ short specifier ] );
```

Syntax for TAL Programmers

```
error := PROCESS_ACTIVATE_ ( processhandle      ! i
                           ,[ specifier ] );    ! i
```

Parameters

error returned value

INT

indicates the outcome of the operation. It returns one of these values:

0	Process successfully activated.
2	Process is already in the ready state.
11	Process does not exist.
48	Security violation.
201	Unable to communicate with processor where the process is running.

processhandle input

INT .EXT:ref:10

is a process handle that specifies the process to be activated:

- To activate a single process, specify the process handle of that process.
- To activate a process pair, specify the process handle of either the primary or backup process.

specifier input

INT:value

for a named process pair, indicates whether both members should be activated. Valid values are:

0 Activate the specified process.

1 Activate both members of current instance of named process pair if the specified process is part of a named process pair; otherwise, activate the specified process.

If this parameter is omitted, 0 is used.

Considerations

- Procedure use

  You can use PROCESS_ACTIVATE_ to activate any suspended process or process pair, even if it was suspended by a call to SUSPENDPROCESS.

- Security

  When PROCESS_ACTIVATE_ is called on a Guardian process, the caller must be the super ID, the group manager of the process access ID, or a process with the same process access ID as the process or process pair being activated. For information about the process access ID, see the description under General Considerations on page 12-62 and the *Guardian User's Guide*.

  The caller must be local to the same system as the specified process. A process is considered to be local to the system on which its creator is local. A process is considered to be remote, even if it is running on the local system, if its creator is remote. (In the same manner, a process running on the local system whose creator is also running on the local system might still be considered remote because it's creator's creator is remote.)

  A remote process running on the local system can become a local process by successfully logging on to the local system with a call to the USER_AUTHENTICATE_ (or VERIFYUSER) procedure. After a process logs on to the local system, any processes that it creates are considered local.

  When PROCESS_ACTIVATE_ is called on an OSS process, the security rules that apply are the same as those that apply when the OSS `kill()` function is called. See the `kill(2)` function reference pages either online or in the *Open System Services System Calls Reference Manual* for details.

Safeguard Considerations

For information on processes protected by Safeguard, see the *Safeguard Reference Manual*.

OSS Considerations

When used on an OSS process, PROCESS_ACTIVATE_ has the same effect as calling the OSS `kill()` function with the input parameters as follows:

- The *`signal`* parameter set to `SIGCONT`
- The *`pid`* parameter set to the OSS process ID of the process identified by *`process-handle`* in the PROCESS_ACTIVATE_ call.

The SIGCONT signal is delivered to the target process.

Related Programming Manual

For programming information about the PROCESS_ACTIVATE_ procedure, see the *Guardian Programmer's Guide*.

PROCESS_CREATE_ Procedure (Superseded by PROCESS_LAUNCH_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The PROCESS_CREATE_ procedure creates a new process and, optionally, assigns a number of process attributes.

You can use this procedure to create only Guardian processes, although you can call it from a Guardian process or an OSS process. The program file must contain a program for execution in the Guardian environment. The program file and user library file must reside in the Guardian name space; that is, they must not be OSS files.

You can specify that the new process be created in either a waited or nowait manner. When it is created in a waited manner, identification for the new process is returned directly to the caller. When it is created in a nowait manner, its identification is returned in a system message sent to the caller's $RECEIVE file.

DEFINEs can be propagated to the new process. The DEFINEs can come from the caller's context or from a buffer of DEFINEs saved by the DEFINESAVE procedure.

Any of the file-name parameters can contain DEFINE names.

Syntax for C Programmers

```
#include <cextdecs(PROCESS_CREATE_)>

short PROCESS_CREATE_ ( [ const char *program-file ]/* i,i1*/
                     ,[ short length ]                /* i,i1*/
                     ,[ const char *library-file ]    /*i,i 2*/
                     ,[ short length ]                /* i,i2*/
                     ,[ const char *swap-file ]       /* i,i3*/
                     ,[ short length ]                /* i,i3*/
                     ,[ const char *ext-swap-file ]   /* i,i4*/
                     ,[ short length ]                /* i,i4*/
                     ,[ short priority ]               /* i 5*/
                     ,[ short processor ]              /* i 6*/
                     ,[ short *processhandle ]         /* o 7*/
                     ,[ short *error-detail ]          /* o 8*/
                     ,[ short name-option ]            /* i 9*/
                     ,[ const char *name ]            /*i,i10*/
                     ,[ short length ]                /*i,i10*/
                     ,[ char *process-descr ]         /*o,i11*/
                     ,[ short maxlen ]               /* o,i11*/
                     ,[ short *process-descr-len ]    /* o 12*/
                ,[ __int32_t nowait-tag ]             /* i 13*/
                     ,[ const char *hometerm ]       /*i,i 14*/
                     ,[ short length ]               /* i,i14*/
                     ,[ short memory-pages ]          /* i 15*/
                     ,[ short jobid ]                 /* i 16*/
                     ,[ short create-options ]        /* i 17*/
                     ,[ const char *defines ]        /*i,i 18*/
                     ,[ short length ]               /*i,i 18*/
                     ,[ short debug-options ]         /* i 19*/
                ,[ __int32_t pfs-size ] );          /* i 20 */;
```

- CEXTDECS (through the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.
- The parameter *maxlen* specifies the maximum length in bytes of the character string pointed to by *process-descr*, the actual length of which is returned by *process-descr-len*. All three of these parameters must either be supplied or be absent.
- Some character-string parameters to PROCESS_CREATE_ are followed by a parameter *length* that specifies the length in bytes of the character string. Where the parameters are optional, the character-string parameter and the corresponding length parameter must either both be supplied or both be absent.

Syntax for TAL Programmers

```
error := PROCESS_CREATE_ ( [ program-file: length ] ! i,i 1 !
                         ,[ library-file: length ] ! i:i 2 !
                         ,[ swap-file: length ]    ! i:i 3 !
                         ,[ ext-swap-file: length  ! i:i 4 !
                         ,[ priority ]               ! i 5 !
                         ,[ processor ]              ! i 6 !
                         ,[ processhandle ]          ! o 7 !
                         ,[ error-detail ]           ! o 8 !
                         ,[ name-option ]            ! I 9 !
                         ,[ name: length ]         ! i:i10 !
                         ,[ process-descr: maxlen ]!o:i 11 !
                         ,[ process-descr-len ]     ! o 12 !
                         ,[ nowait-tag ]            ! i 13 !
                         ,[ hometerm: length ]     !i:i 14 !
                         ,[ memory-pages ]          ! i 15 !
                         ,[ jobid ]                 ! i 16 !
                         ,[ create-options ]        ! i 17 !
                         ,[ defines: length ]      !i:i 18 !
                         ,[ debug-options ]         ! i 19 !
                         ,[ pfs-size ] );           ! i 20 !
```

Parameters

error returned value

INT

indicates the outcome of the operation. Table 12-3 on page 12-111 summarizes possible values for *error*.

program-file:*length* input:input

STRING .EXT:ref:*, INT:value

if supplied and if *length* is not 0, specifies the name of the program file to be run. If used, the value of *program-file* must be a valid file name and must be exactly *length* bytes long. The file must reside in the Guardian name space and must contain a program for execution in the Guardian environment.

The new process is created on the node where the program file resides. If the program file name is partially qualified, it is resolved using the =_DEFAULTS DEFINE. If you specify a file on the subvolume $SYSTEM.SYSTEM and the file is not found, PROCESS_CREATE_ then searches on the subvolume $SYSTEM.SYS*nn*.

For a description of file name syntax, see Appendix D, File Names and Process Identifiers.

This parameter must be supplied unless the caller is creating its backup process.

library-file:*length* input:input

STRING .EXT:ref:*, INT:value

if supplied and if *length* is not 0 or -1, specifies the name of the user library file to be used by the process. If used, the value of *library-file* must be exactly *length* bytes long. If the library file name is partially qualified, it is resolved using the =_DEFAULTS DEFINE. The user library file must be on the same node as the process being created and must reside in the Guardian name space.

If *library-file* is specified, unresolved external references are resolved first from the specified *library-file*, then from the system library.

If *library-file* is specified and *length* is -1, the new process is to run with no user library file. (The references that were previously resolved from the user library are resolved from the system library.) For the program to remove a linkage to a library file, the caller must have write permission to the program file.

If *library-file* is not specified or *length* is 0, then the program runs with the same library file as the last time it was run (or with no file if that was how it was run) or with the library file currently executing. Write permission to the program file is not required. For more information about TNS user libraries, see the *Binder Manual*. For more information about TNS/R native user libraries and shared run-time libraries, see the *nld and noft Manual*. For more information about dynamic-link libraries (including native user libraries used with PIC programs), see the *ld and rld Reference Manual*.

If an external reference cannot be resolved, it is modified to invoke the debugger when referenced. PROCESS_CREATE_ then returns a warning 14 and issues a warning message to the home terminal the first time the program is run. (The warning 14 and the terminal message are issued again the first time the program is run following a system load).

swap-file:*length* input:input

STRING .EXT:ref:*, INT:value

is not used, but you can provide it for informational purposes. If supplied, the swap file must be on the same system as the process being created. If the supplied name is in local form, the system where the process is created is assumed. Processes swap to a file that is managed by the Kernel-Managed Swap Facility (KMSF). For more information on this facility, see the *Kernel-Managed Swap Facility (KMSF) Manual*. To reserve swap space for the process, create the process using the PROCESS_LAUNCH_ procedure and specify the Z^SPACE^GUARANTEE field of the *param-list* parameter. Alternatively, use the `nld` utility to set TNS/R native process attributes or the `eld` utility to set TNS/E native processes.

For TNS processes on RVUs preceding the D42 RVU, if supplied and if *length* is not 0, *swap-file* specifies the name of a file to be used as the swap file for the user data stack segment of the process. If used, the value of *swap-file* must be

exactly *length* bytes long. If the swap file name is partially qualified, it is resolved using the =_DEFAULTS DEFINE. The swap file must be on the same node as the process being created and must be an unstructured file.

See “Considerations” for more information about swap files.

ext-swap-file:*length* input:input

STRING .EXT:ref:*, INT:value

for TNS processes, if not specified or *length* is 0, the Kernel-Managed Swap Facility (KMSF) allocates swap space for the default extended data segment of the process. For more information on this facility, see the *Kernel-Managed Swap Facility (KMSF) Manual*.

For TNS processes, if specified and *length* is not 0, *ext-swap-file* contains the name of a file to be used as the swap file for the default extended data segment of the process. If used, the value of *ext-swap-file* must be exactly *length* bytes long. If the swap file name is partially qualified, it is resolved using the =_DEFAULTS DEFINE. The swap file must be on the same node as the process being created and must be an unstructured file.

For native processes, this parameter is ignored because native processes do not need an extended swap file.

See “Considerations” for more information about swap files.

priority input

INT:value

is the initial execution priority to be assigned to the new process. Execution priority is a value in the range of 1 to 199, where 199 is the highest possible priority. If you omit this parameter, or if you specify -1, the priority of the caller is used. If you specify 0, a value less than -1, or a value greater than 199, error 2 is returned.

processor input

INT:value

specifies the processor in which the new process is to run. If you omit this parameter, or if you specify -1, the processor is chosen as follows:

Backup process:	determined by system
Other process on local system:	same processor as caller
Process on remote system:	determined by system

The processor number can be obtained by passing *processhandle* to PROCESSHANDLE_DECOMPOSE_.

processhandle output

INT .EXT:ref:10

returns the process handle of the new process. If you created the process in a nowait manner, the process handle is returned in the completion message sent to $RECEIVE rather than through this parameter.

error-detail output

INT .EXT:ref:1

returns additional information about some classes of errors. The sets of values for *error-detail* vary according to the *error* value, as described in Table 12-4 on page 12-120.

name-option input

INT:value

specifies whether the process is to be named and, if so, whether the caller is supplying the name or the system must generate it. Valid values are:

0 Process is unnamed (unless the RUNNAMED object file attribute is set for the program file).

1 Process is named; name is supplied in *name*.

2 Process is named; system must generate a name. (The generated name is four characters long, not including the $.)

3 Process is caller's backup; use caller's name.

4 Process is named; system must generate a name. (The generated name is five characters long, not including the $.

If this parameter is omitted, 0 is used.

If either the program file or the library file (if any) has the RUNNAMED program-file flag set, the system overrides *name-option* of 0 and generates a name. The system also generates a name if RUNNAMED is set and *name-option* is 2, 4, or omitted. The generated name is four characters long, not including the $, unless *name-option* is 4. In the latter case, the name is five characters long, not including the $.

name:*length* input:input

STRING .EXT:ref:*, INT:value

if *name-option* is 1 and *length* is not 0, specifies a name to be assigned to the new process. If used, the value of *name* must be exactly *length* bytes long. The name can include a node name, but the node must match that of the program file. See "General Considerations," later in this subsection, and Appendix B, Reserved Process Names for information about reserved process names.

For other values of *name-option*, this parameter should be omitted (or *length* should be set to 0), since the system will either generate a name or, in the case of backup creation, use the name of the caller.

process-descr:*maxlen* output:input

STRING .EXT:ref:*, INT:value

if present and *maxlen* is not 0, returns a process descriptor suitable for passing to FILE_OPEN_. *maxlen* specifies the length of the string variable *process-descr* in bytes. If it is not 0, the value of *maxlen* must be at least 33.

If you created the process in a nowait manner, the process descriptor is returned in the completion message sent to $RECEIVE rather than in *process-descr*.

process-descr-len output

INT .EXT:ref:1

if *process-descr* is returned, contains its actual length in bytes.

nowait-tag input

INT(32):value

if present and not -1D, indicates that the process is to be created in a nowait manner; the procedure returns as soon as process creation is initiated. For details, see Nowait Considerations on page 12-46.

If *nowait-tag* is -1D or omitted, the process is created in a waited manner.

hometerm:*length* input:input

STRING .EXT:ref:*, INT:value

if supplied and if *length* is not 0, is a file name that designates the home terminal for the new process. If used, the value of *hometerm* must be exactly *length* bytes long. If *hometerm* is partially qualified, it is resolved using the =_DEFAULTS DEFINE.

hometerm can be a named or unnamed process. The default value is the home terminal of the caller.

memory-pages input

INT:value

For TNS processes, specifies the minimum number of 2048-byte memory pages allocated to the new process for user data. The actual amount of memory allocated is processor-dependent. If *memory-pages* is either omitted or less than the value previously assigned either by a compiler directive at compile time or by a Binder command at bind time, the previously assigned value is used. In any case, the maximum number of pages permitted is 64.

For native processes, this parameter is ignored. To specify the maximum size of the main stack, create a new process using the PROCESS_LAUNCH_ procedure and specify the Z^MAINSTACK^MAX field of the *param-list* parameter. Alternatively, use the `nld` utility to set the TNS/R process attributes or the `eld` utility to set the TNS/E process attributes .

jobid input

INT:value

if present and not 0 or -1, is an integer (job ID) that identifies the job to be created. The new process is the first process of the job and the caller is the job ancestor of the new process. This value is used by the NetBatch scheduler. See “Batch Processing Considerations” for information about how to use this parameter.

create-options input

INT:value

provides information about the environment of the new process. The fields are:

<9>	= 0	If the caller is named, the process deletion message, if any, will go only to the current instance of the calling process.
	= 1	If the caller is named, the process deletion message, if any, will go to whatever process has the calling process’s name (regardless of sequence number) at that time.
<10>	= 0	Force new process into a low PIN if the calling process has the inherited force-low attribute set.
	= 1	Ignore the value of the caller’s inherited force-low attribute.
<11:12>	= 0	Propagate only the DEFINEs in the caller’s context.
	= 1	Propagate only the DEFINES in the *defines* parameter.
	= 2	Propagate both sets of defines; in case of name conflicts, use the ones in *defines*.
<13>	= 0	Use caller’s DEFINE mode.
	= 1	Use value in bit 14.
<14>	= 0	DEFINEs disabled (ignored if bit 13 is 0).

	= 1	DEFINEs enabled (ignored if bit 13 is 0).
<15>	= 0	Can run at any PIN.
	= 1	Requires low PIN (in range 0 through 254).

The default value is 0.

If you specify *create-options*.<9> = 1, the process deletion message (in the event that the created process terminates) is sent to any process that has the calling process's name at that time, regardless of the sequence number. If you specify *create-options*.<9> = 0, the process deletion message is sent only to the instance of the process or process pair to which the calling process belongs. An "instance" is any process in an unbroken chain of primary and backup processes. Every process that is part of an instance has the same sequence number.

If you specify *create-options*.<15> = 1 (requires low PIN), the program is run at a low PIN. If you specify *create-options*.<15> = 0 (can be assigned any PIN), the program runs at a PIN of 256 or higher if its program file and library file (if any) have the HIGHPIN program-file flag set and if a high PIN is available. However, if the calling process has the inherited force-low attribute set and you specify "can run at any PIN," the new process is forced into a low PIN even if all of the other conditions for running at a high PIN are met. See "Compatibility Considerations" and "DEFINE Considerations" for more information.

defines:*length* input:input

STRING .EXT:ref:*, INT:value

if supplied and if *length* is not 0, specifies a set of DEFINEs to be propagated to the new process. The value of *defines* must be exactly *length* bytes long. The set of DEFINEs should have been created through one or more calls to DEFINESAVE. For all cases except backup creation, DEFINEs are propagated according to the values specified in *create-options*. See "DEFINE Considerations" for details.

When a process creates its backup, all of the caller's DEFINEs are propagated regardless of *create-option*. If *defines* is specified, it is ignored.

debug-options input

INT:value

sets the debugging attributes for the new process. The fields are:

<0:11>		Reserved (specify 0)
<12>	1	Enter Debug or the Inspect debugger at the first executable instruction of the program's MAIN procedure.
	0	Begin normal program execution.
<13>	1	If the process traps, create a saveabend file.
	0	If the process traps, do not create a saveabend file.
<14>	1	Use debugger specified in bit 15 and saveabend option specified in bit 13 regardless of program-file flag setting.
	0	Use standard rules for debugger selection.
<15>	1	Use the Inspect debugger.
	0	Use Debug.

When *debug-options* is specified, bits 13 and 15 are ORed with the corresponding flags in the program file. If the result is that bit 13 is set but bit 15 is not, then 15 is also turned on (that is, if “save file creation” is selected, the Inspect debugger becomes the selected debugger).

If bit 14 is set, the above paragraph on debugger selection does not apply. The debugger specified by bit 15 is used, regardless of the flags in the program file.

If *debug-options* is omitted, then the debugger and saveabend defaults are set from the flags in the program file (set either by compiler directives at compile time, `nld` flag at link time, or Binder command at bind time) after these flags are ORed with the corresponding states of the calling process.

pfs-size input

INT(32):value

meaningful only if the process is being created on a pre-G06 RVU. On G06 and later RVUs, this value is range checked, but is otherwise ignored.

If present and nonzero, this parameter specifies the size in bytes of the process file segment (PFS) of the new process. In G-series RVUs, maximum PFS size is 8 MB. In H-series RVUs, maximum PFS size is 32 MB. A value in this range overrides the `nld` or Binder value stored in the program file. If you omit *pfs-size* or specify 0:

- the `nld` or Binder value is used if it is nonzero
- a default value is used otherwise

General Considerations

- Partially qualified file names are resolved using the contents of the caller’s =_DEFAULTS DEFINE. If a node name is not present in either the file name or the

appropriate attribute of the DEFINE, the resolved name will include the caller's node.

See below for details on resolution of specific file-name parameters.

- For TNS and accelerated processes on RVUs preceding the D42.00 RVU, if *swap-file* or *ext-swap-file*:
  - is specified and a file with that name exists, that file is used for memory swapping of the user data stack (*swap-file*) or the default extended data segment (*ext-swap-file*) during execution of the process; if no file of that name exists, then a file of that name and of the necessary size is created and used for swapping. If the file name is partially qualified, the system uses the =_DEFAULTS DEFINE to resolve it.
  - Specifies the name of a temporary file that is already in use, an error is returned.
  - Specifies only the disk volume name, then a temporary file is created on the specified disk device.
  - Is not specified or *length* is 0, then the SWAP volume name in the =_DEFAULTS DEFINE is used if available. Otherwise, the system chooses where to place the file.
- Creation of the backup of a named process pair

  If the backup of a named process pair is created, the backup process becomes the creator or mom of the primary (that is, of the caller to PROCESS_CREATE_) and the primary becomes the mom of the newly created backup process. See the discussions of "mom process" and "ancestor process" in the *Guardian Programmer's Guide*.
- Program file and user library file differences

  A user library is an program file containing one or more procedures. The difference between a program file and a library file is that the library file cannot contain a MAIN procedure; a program file must contain a MAIN procedure. Undefined external references in a program file are resolved from the user library, if any, or the system library. Unresolved references in a library are resolved only from the system library.
- Library conflict—PROCESS_CREATE_ error

  The library file for a process can be shared by any number of processes. However, when a program is shared by two or more processes, all processes must have the same user library configuration; that is, all processes sharing the program either have the same user library, or they have no user library. A library conflict error occurs when there is already a copy of the program running with a library configuration different from that specified in the call to PROCESS_CREATE_.

- Device subtypes for named processes (process subtypes)

  The device subtype (or process subtype) is a program file attribute that can be set by a TAL compiler directive at compile time, `nld` flag at link time, or Binder command at bind time. You can obtain the device type and subtype of a named process by calling FILE_GETINFO[BYNAME]_ , FILEINFO, or DEVICEINFO.

  Note that a process with a device subtype other than 0 must always be named.

  There are 64 process subtypes available, where 0 is the default subtype for general use. The other subtypes are as follows:

  1 to 47 are reserved for definition by HP. Currently, 1 is a CMI process, 2 is a security monitor process, 30 is a device simulation process, and 31 is a spooler collector process. Also, for subtypes 1 to 15, PROCESS_CREATE_ rejects the create request with an invalid process subtype error unless the caller has a creator access ID of the super ID, or the program file is licensed, or the program file has the PROGID attribute set and an owner of the super ID.

  48 to 63 are for general use. Any user can create a named process with a device subtype in this range.

  For a list of all device types and subtypes, see Appendix A, Device Types and Subtypes.

- Reserved process names

  The operating system reserved process name space includes these names: $X*name*, $Y*name*, and $Z*name*, where *name* is from 1 through 4 alphanumeric characters. You should not use names of this form in any application. System-generated process names (from PROCESS_LAUNCH_, PROCESS_SPAWN_, PROCESS_CREATE_ , NEWPROCESS[NOWAIT], PROCESSNAME_CREATE_ , CREATEPROCESSNAME and CREATEREMOTENAME procedures) are selected from this set of names. For more information about reserved process names. see Appendix B, Reserved Process Names.

- Creator access ID (CAID) and process access ID (PAID)

  The creator access ID of the new process is always the same as the process access ID of the creator process. The process access ID of the new process is the same as that of the creator process unless the program file has the PROGID attribute set; in that case the process access ID of the new process is the same as the user ID of the program file's owner and the new process is always local.

- I/O error to the home terminal

  An I/O error to the home terminal can occur if there are undefined externals in the program file and PROCESS_CREATE_ is unable to open or write to the home terminal to display the undefined externals messages. The *error-detail* parameter contains the file-system error number that resulted from the open or write that failed.

Nowait Considerations

- If you call this procedure in a nowait manner, the results are returned in the nowait PROCESS_LAUNCH_ or PROCESS_CREATE_ completion message (-102), not the output parameters of the procedure. The format of this completion message is described in the *Guardian Procedure Errors and Messages Manual*. If *error* is not 0, no completion message is sent to $RECEIVE. Errors can be reported either on return from the procedure, in which case *error* and *error-detail* might be meaningful, or through the completion message sent to $RECEIVE.

Compatibility Considerations

- If the new process is to be accessible to a process with a low PIN, then it must be forced into a low PIN (below 255). You can force the process into a low PIN either by specifying *create-options*.<15> = 1 (requires a low PIN), or by making sure that the program-file high-PIN flag is off.
- If you want the new process to be forced into a low PIN only if the calling process was forced into a low PIN, specify *create-options*.<10> = 0 (requires a low PIN if the caller has the inherited force-low attribute set) and *create-options*.<15> = 0 (can have any PIN).
- If you want explicit control over each child process with respect to running with a high or low PIN, specify *create-options*.<10> = 1 (ignore the caller's inherited force-low attribute) and *create-options*.<15> = either 1 (requires a low PIN) or 0 (can be assigned any PIN) as appropriate.
- If the new process is unnamed, it must be forced into a low PIN if it is to be accessible to processes that do not know about high PINs.
- If the new process has a high PIN and has a name with five or fewer characters (not counting the $), it is accessible to any high PIN process running on any node in the network.
- For further information on compatibility, see the *Guardian Programmer's Guide* and the *Guardian Application Conversion Guide*.

DEFINE Considerations

- DEFINEs are propagated to the new process from the process context of the caller, from a caller-supplied buffer containing DEFINEs collected by calls to DEFINESAVE, or from both of these. DEFINEs are propagated to the new process according to the DEFINE mode of the new process and the propagation option specified in *create-options*. If both sets of DEFINEs are propagated and both sets contain a DEFINE with the same name, the DEFINE in the caller-supplied buffer is used. When a caller is creating its backup, the caller's DEFINEs are always propagated, regardless of the options chosen.

  The =_DEFAULTS DEFINE is always propagated, regardless of the options chosen. If the DEFINE buffer contains a =_DEFAULTS DEFINE, that one is

propagated; otherwise, the =_DEFAULTS DEFINE in the caller's context is propagated.

Buffer space for DEFINEs being propagated to a new process is limited to 2 MB whether the process is local or remote. However, the caller can propagate only as many DEFINEs as the child's PFS can accommodate in the buffer space for the DEFINEs themselves and in the operational buffer space needed to do the propagation. The maximum number of DEFINEs that can be propagated varies depending upon the size of the DEFINEs being passed.

- When a process is created, its DEFINE working set is initialized with the default attributes of CLASS MAP.
- The *program-file*, *library-file*, *swap-file*, or *ext-swap-file* can be DEFINE names; PROCESS_CREATE_ uses the disk volume or file given in the DEFINE. If *program-file* is a DEFINE name but no such DEFINE exists, an error is returned. If any of the other names is a DEFINE name but no such DEFINE exists, the procedure behaves as if no name were specified. This feature of accepting names of nonexistent DEFINEs as input gives the programmer a convenient mechanism that allows, but does not require, user specification of the location of the library file, the swap file, or the extended swap file.
- For each process, a count is kept of the changes to that process's DEFINEs. This count is always 0 for newly-created processes. The count is incremented each time the procedures DEFINEADD, DEFINEDELETE, DEFINESETMODE, and DEFINEDELETEALL are invoked and a consequent change to the process context occurs. In the case of DEFINEDELETE and DEFINEDELETEALL, the count is incremented by one even if more than one DEFINE is deleted. The count is also incremented if the DEFINE mode of the process is changed. If a call to CHECKDEFINE causes a DEFINE in the backup to be altered, deleted, or added, then the count for the backup process is incremented.

Batch Processing Considerations

Note. The job ancestor facility is intended for use by the NetBatch product. Other applications that use this facility might be incompatible with the NetBatch product.

- When the process being created is part of a batch job, PROCESS_CREATE_ sends a job process creation message to the job ancestor of the batch job. (See the discussion of "job ancestor" in the *Guardian Programmer's Guide*.) The message identifies the new process and contains the job ID as originally assigned by the job ancestor. This enables the job ancestor to keep track of all the processes belonging to a given job.

  For the format of the job process creation message, see the *Guardian Procedure Errors and Messages Manual*.
- PROCESS_CREATE_ can create a new process and establish that process as a member of the caller's batch job. In that case the caller's job ID is propagated to

the new process. If the caller is part of a batch job, to start a new process that is part of the caller's batch job, omit *jobid* or set it to -1.

- PROCESS_CREATE_ can create a new process separate from any batch job, even if the caller is a process that belongs to a batch job. In that case the job ID of the new process is 0. To start a new process that is not part of a batch job, specify 0 for *jobid*.
- PROCESS_CREATE_ can create a new batch job and establish the new process as a member of the newly created batch job. In that case, the caller becomes the job ancestor of the new job; the job ID supplied by the caller becomes the job ID of the new process. To start a new batch job, specify a nonzero value (other than -1) for *jobid*.

  A job ancestor must not have a process name that is greater than four characters (not counting the dollar sign). When the caller of PROCESS_CREATE_ is to become a job ancestor, it must conform to this requirement.
- When *jobid* is omitted or set to -1:
  - If the caller is not part of a batch job, neither is the newly created process; its job ID is 0.
  - If the caller is part of a batch job, the newly created process is part of the same job because its job ID is propagated to the new process.
- Once a process belongs to a batch job, it remains part of the job.

Safeguard Considerations

For information on processes protected by Safeguard, see the *Safeguard Reference Manual*.

OSS Considerations

- You cannot create an OSS process using the PROCESS_CREATE_ procedure. PROCESS_CREATE_ returns error 12 if you try. Use the PROCESS_SPAWN_ procedure or OSS functions to create an OSS process.
- You can call PROCESS_CREATE_ from an OSS process to create a Guardian process.
- Every Guardian process has these security-related attributes for accessing OSS objects. These attributes are passed, unchanged, from the caller to the new process, whether the caller is an OSS process or a Guardian process:
  - Real, effective, and saved user ID
  - Real, effective, and saved group ID
  - Group list
  - Login name

- Current working directory (cwd)
- Maximum file size
- Default OSS file security

  No other OSS process attribute is inherited by the new process.

- OSS file opens in the calling process are not propagated to the new process.

Example

```
err := PROCESS_CREATE_ ( pfile^name , , , , , , proc^handle,
                         error^detail );
```

Related Programming Manuals

For programming information about the PROCESS_CREATE_ procedure, see the *Guardian Programmer's Guide*. For programming information on batch processing, see the appropriate NetBatch manual.

PROCESS_DEBUG_ Procedure

Summary

The PROCESS_DEBUG_ procedure invokes the debugging facility on the calling process or on another process.

The operating system provides a debugging facility that responds to debug events by passing control to one of two debugging utilities: Debug or Inspect. Debug is a low-level debugger. Inspect is an interactive symbolic debugger that lets you control program execution, display values, and modify values in terms of source-language symbols.

Syntax for C Programmers

```
#include <cextdecs(PROCESS_DEBUG_)>

short PROCESS_DEBUG_ ( [ short *processhandle ]
                      ,[ const char *terminal-name ]
                      ,[ short length ]
                      ,[ short now ] );
```

The parameter *length* specifies the length in bytes of the character string pointed to by *terminal-name*. The parameters *terminal-name* and *length* must either both be supplied or both be absent.

Syntax for TAL Programmers

```
error := PROCESS_DEBUG_ ( [ processhandle ]            ! i
                         ,[ terminal-name:length ]     ! i:i
                         ,[ now ] );                   ! i
```

Parameters

error returned value

INT

returns a file-system error number indicating the outcome of the process debug attempt. Possible values include:

0	Debug request accepted. If the process to be debugged is not the calling process, the request might have been queued.
11	Process does not exist.
48	Security violation.
201	Unable to communicate with the processor of the process.
640	The target process runs in privileged mode and the *now* parameter was not set equal to 1.

processhandle input

INT .EXT:ref:10

is the process handle of the process to be debugged. If *processhandle* is omitted or null, the calling process is to be debugged. The null process handle is one which has -1 in each word (Refer to Guardian procedure call, PROCESSHANDLE_NULLIT_). However, PROCESS_DEBUG also treats a process handle with -1 in the first word as a null process handle.

terminal-name:*length* input:input

STRING .EXT:ref:*, INT:value

if supplied and if *length* is not 0, is a file name that designates the terminal from which the process is to be debugged. If used, the value of *terminal-name* must be exactly *length* bytes long. If *terminal-name* is partially qualified, it is resolved using the contents of the =_DEFAULTS DEFINE.

The default is the current home terminal of the process to be debugged.

now input

INT:value

if 1, specifies that the process be debugged immediately (even if it is currently executing privileged code); if omitted or 0, specifies that the normal debug sequence be executed.

The process access ID (PAID) of the calling process must be the super ID to use this parameter.

If the calling process runs only in privileged mode, *now* must be set to 1 or an error is returned.

Considerations

- The caller of PROCESS_DEBUG_ must be the super ID, the group manager of the process access ID, or a process with the same process access ID as the specified process or process pair. For information about the process access ID, see General Considerations on page 12-62 and the *Guardian User's Guide*.

  The caller must be local to the same system as the specified process. A process is considered to be local to the system on which its creator is local. A process is considered to be remote, even if it is running on the local system, if its creator is remote. (In the same manner, a process running on the local system whose creator is also running on the local system might still be considered remote because it's creator's creator is remote.)

  A remote process running on the local system can become a local process by successfully logging on to the local system with a call to the USER_AUTHENTICATE_ procedure (or VERIFYUSER). After a process logs on to the local system, any processes that it creates are considered local.

- While a process is in the debug state, you can interactively display and modify the contents of its registers and data area, and set breakpoints. To debug a program, you must have execute access to run the program and read access to the program file.

- In addition to placing an explicit call to the PROCESS_DEBUG_ (or DEBUG) procedure in the source program, you can force a process into the debug state by:

  - Starting the process using the TACL RUND (RUN DEBUG) command. The process enters the debug state before the first instruction of the MAIN procedure executes.
  - Starting the process with a call to PROCESS_CREATE_, PROCESS_SPAWN_, NEWPROCESS, NEWPROCESSNOWAIT, OSS `tdm_fork()`, OSS `tdm_spawn()`, or one of the OSS `tdm_exec` set of functions, and setting the appropriate option. The process enters the debug state before the first instruction of the MAIN procedure executes.
  - Starting the process from the command interpreter. While the process is executing, press the BREAK key. The command interpreter returns to the command input mode. Find the *cpu,pin* of the process and type in DEBUG *cpu,pin*, or find the name of the process (if it is named) and type in DEBUG *process-name*.
  - Issuing a DEBUG command to a command interpreter on another local terminal while the process is executing. Find the *cpu,pin* of the process and type in DEBUG *cpu,pin*, or find the name of the process (if it is named) and type in DEBUG *process-name*.
  - Specifying a breakpoint when a process is in the debug state. When the breakpoint is reached, the process enters the debug state.
  - Issuing a DEBUG command to a TACL process running at another local terminal.
- You can use Inspect by setting the Inspect attribute associated with a process. The value of a process's Inspect attribute can be set with:
  - The ?INSPECT or ?SAVEABEND TAL compiler directive
  - The `nld` -SET INSPECT or -SET SAVEABEND flags during a linking session
  - The Binder SET INSPECT or SET SAVEABEND commands during a binding session
  - The TACL SET INSPECT command before the RUN command that starts the process
  - The INSPECT parameter of the RUN command that starts the process
  - The appropriate option in the PROCESS_CREATE_, PROCESS_SPAWN_, NEWPROCESS, NEWPROCESSNOWAIT, OSS `tdm_fork()`, OSS `tdm_spawn()`, or one of the OSS `tdm_exec` set of functions.
- Processes inherit the Inspect attribute from their ancestor processes.
- Calling PROCESS_DEBUG_ and passing no parameters (or specifying only the caller's process handle) is not the exact equivalent of calling the DEBUG procedure. Some processes (in particular, system processes) would need to specify the *now* parameter as equal to 1 (the default is 0). DEBUG, which has no *now* parameter, functions as if it had a *now* parameter set equal to 1.

In general, the preferred method for a process to invoke the debug facility on itself is to call DEBUG rather than to call PROCESS_DEBUG_.

OSS Considerations

When used on an OSS process, PROCESS_DEBUG_ forces the process into the Inspect debugger. You can change the home terminal by specifying *terminal-name*. Note that the home terminal is often the same device as the OSS controlling terminal.

To debug an OSS process, one of these must be true:

- The calling process must have the appropriate privileges; that is, it must be locally authenticated as the super ID on the system where the target process is executing.
- All these apply:
  - The caller's effective user ID is the same as the saved user ID of the target process.
  - The caller has sufficient "non-remoteness": that is, the caller is locally authenticated, or the target process is remotely authenticated and the caller is authenticated from the viewpoint of the system where the target process is executing.
  - The caller has read access to the program file and any library files.
  - The program does not contain PRIV or CALLABLE routines.
  - The target is not a system process.
  - The *now* parameter is not specified.

Only program file owners and users with appropriate privileges are able to debug programs that set the user ID.

Example

```
error := PROCESS_DEBUG_ ( proc^handle, terminal:length );
```

Related Programming Manual

For information about the Debug facility, see the *Debug Manual*. For information about the Inspect facility, see the *Inspect Manual*. For programming information about the PROCESS_DEBUG_ procedure, see the *Guardian Programmer's Guide*.

PROCESS_DELAY_ Procedure (H-Series RVUs Only)

Summary

The PROCESS_DELAY_ procedure permits a process to suspend itself for a timed interval.

Syntax for C Programmers

```
#include <cextdecs(PROCESS_DELAY_)>

void PROCESS_DELAY_ ( long long (Timeout );
```

Syntax for TAL Programmers

```
CALL PROCESS_DELAY_( timeout ) ;          ! i
```

Parameters

timeout input

FIXED:value

specifies the time, in microsecond units, that the caller of PROCESS_DELAY_ is to be suspended.

Considerations

- The process stops executing for at least timeout microseconds.
- *time-period* value <= 0D

  A value of less than or equal to 0D results in no delay as such, but returns this process to the Ready List.
- Measuring time by the processor clock

  The PROCESS_DELAY_ procedure measures time according to the internal clock of the processor in which the calling process is executing. Processor time is time as measured by a particular process. Typically, processor time differs slightly from

system time and varies slightly from processor to processor. System time is determined by taking the average of all the processor times in the system.

When measuring short intervals, the difference between processor time and system time is negligible. However, when measuring long intervals (such as several hours or more), the difference can be noticeable. For a discussion about measuring long time intervals, see Considerations on page 14-142.

Note. Currently, the minimum *timeout* supported for PROCESS_DELAY_ procedures is:

- 2 milliseconds for H06.12 and earlier H-series RVUs
- 1 millisecond for privileged processes from H06.13 onwards
- 2 milliseconds for non-privileged processes from H06.13 onwards

---

Example

```
CALL PROCESS_DELAY_(60000000F); -- delay for 60 seconds
```

PROCESS_GETINFO_ Procedure

Summary

The PROCESS_GETINFO_ procedure obtains a limited set of information about a specified process.

A related procedure, PROCESS_GETINFOLIST_, obtains detailed information about a particular process or set of processes that meet specified criteria.

Syntax for C Programmers

Note. In the TNS/E environment, the CEXTDECS file uses the *int* data type for 32-bit values. This is a change from the TNS and TNS/R environments where CEXTDECS uses the *long* data type for 32-bit values.

```
#include < cextdecs(PROCESS_GETINFO_)>
                                                  /*input error*/
                                                 /*output detail*/
short PROCESS_GETINFO_ ( [ short *processhandle ] /*i,o   1 */
                       , [ char *proc-fname ]      /*  o     2 */
                       , [ short maxlen ]          /*  o     2 */
                       , [ short *proc-fname-len ] /*  o     3 */
                       , [ short *priority ]       /*  i     4 */
                       , [ short *mom's-processhandle  ] /*i 5 */
                       , [ char *hometerm ]        /*  i     6 */
                       , [ short maxlen ]          /*  i     6 */
                       , [ short *hometerm-len ]   /*  i     7 */
                       , [ long long *process-time ]/* i     8 */
                       , [ short *creator-access-id ]/*i     9 */
                       , [ short *process-access-id ]/*i    10 */
                       , [ short *gmom's-processhandle ]/* i11 */
                       , [ short *jobid ]           /* i    12 */
                       , [ char *program-file ]     /* i    13 */
                       , [ short maxlen ]           /* i    13 */
                       , [ short *program-len ]     /* i    14 */
                       , [ char *swap-file ]        /* i    15 */
                       , [ short maxlen ]           /* i    15 */
                       , [ short *swap-len ]        /* i    16 */
                       , [ short *error-detail ]    /* i    17 */
                       , [ short *proc-type ]       /* i    18 */
                   , [ __int32_t *oss-pid ] );        /* i 19*/;
```

Some character-string parameters to PROCESS_GETINFO_ are followed by a parameter *maxlen* that specifies the maximum length in bytes of the character string and an additional parameter that returns the actual length of the string. Where these parameters are optional, the character-string parameter and the two parameters that follow it must either all be supplied or all be absent.

Syntax for TAL Programmers

```
                                                  ! Input Error
                                               ! ! Output Detail !
error := PROCESS_GETINFO_ ( [ processhandle ]   ! i,o    1    !
                      ,[ proc-fname: maxlen ]   ! o:i    2    !
                      ,[ proc-fname-len ]       ! o      3    !
                      ,[ priority ]             ! o      4    !
                      ,[ mom's-processhandle ]  ! o      5    !
                      ,[ hometerm: maxlen ]     ! o:i    6    !
                      ,[ hometerm-len ]         ! o      7    !
                      ,[ process-time ]         ! o      8    !
                      ,[ creator-access-id ]    ! o      9    !
                      ,[ process-access-id ]    ! o     10    !
                      ,[ gmom's-processhandle ] ! o     11    !
                      ,[ jobid ]                ! o     12    !
                      ,[ program-file: maxlen ]   o:i   13    !
                      ,[ program-len ]          ! o     14    !
                      ,[ swap-file: maxlen ]    ! o:i   15    !
                      ,[ swap-len ]             ! o     16    !
                      ,[ error-detail ]         ! o     17    !
                      ,[ proc-type ]            ! o     18    !
                        ,[ oss-pid ] );            ! o    19!
```

Parameters

error returned value

INT

indicates the outcome of the call. It returns one of these values:

0 Information is returned for the specified process.
1 File-system error; *error-detail* contains the error number.
2 Parameter error; *error-detail* contains the number of the first parameter found to be in error, where 1 designates the first parameter on the left.
3 Bounds error; *error-detail* contains the number of the first parameter found to be in error, where 1 designates the first parameter on the left.
4 Specified process does not exist
5 Unable to communicate with processor
6 Unable to communicate with system

processhandle input,output

INT .EXT:ref:10

specifies the process for which information is to be returned. If *processhandle* is omitted or null, information about the caller is returned. If *processhandle* is null, it returns the caller's process handle. The null process handle is one which has -1 in each word (Refer to Guardian procedure call, PROCESSHANDLE_NULLIT_). However, PROCESS_GETINFO_ also treats a process handle with -1 in the first word as a null process handle.

proc-fname:*maxlen* output:input

STRING .EXT:ref:*, INT:value

if present and *maxlen* is not 0, returns the process file name of the target process. Normally, this value is a process descriptor. However, if the specified process is an I/O process (that is, a process that controls a device or volume), the returned value is the fully qualified device or volume name.

maxlen is the length in bytes of the string variable *proc-fname*.

proc-fname-len output

INT .EXT:ref:1

contains the actual length of the process file name being returned.

priority output

INT .EXT:ref:1

returns the current execution priority of the specified process. The initial priority can be obtained by calling PROCESS_GETINFOLIST_.

mom's-processhandle output

INT .EXT:ref:10

returns the process handle of the mom of the specified process.

hometerm:*maxlen* output:input

STRING .EXT:ref:*, INT:value

if present and *maxlen* is not 0, returns the fully qualified file name of the home terminal of the specified process. *maxlen* specifies the length in bytes of the string variable *hometerm*.

hometerm-len output

INT .EXT:ref:1

contains the actual length in bytes of the value returned in *hometerm*.

process-time output

FIXED .EXT:ref:1

returns the execution time, in microseconds, of the specified process. This value includes processor time consumed by the application code of the process plus processor time consumed by Guardian procedures called.

creator-access-id output

INT .EXT:ref:1

returns the creator access ID of the specified process. See “Considerations” for information about the creator access ID.

process-access-id output

INT .EXT:ref:1

returns the process access ID of the specified process. See “Considerations” for information about the process access ID.

gmom's-processhandle output

INT .EXT:ref:10

returns the process handle of the job ancestor (GMOM) of the specified process.

jobid output

INT .EXT:ref:1

returns the job ID of the specified process. The job ID is a value that was assigned by the job ancestor when the job was created. If *jobid* is 0, the process does not belong to a job.

program-file:*maxlen* output:input

STRING .EXT:ref:*, INT:value

if present and *maxlen* is not 0, returns the fully qualified Guardian program file name of the specified process. *maxlen* specifies the length in bytes of the string variable *program-file*.

program-len output

INT .EXT:ref:1

contains the actual length in bytes of the program file name being returned.

swap-file:*maxlen* output:input

STRING .EXT:ref:*, INT:value

returns $*volume*.#0. Processes do not swap to $*volume*.#0; they swap to a swap file managed by the Kernel-Managed Swap Facility. For more information on this facility, see the *Kernel-Managed Swap Facility (KMSF) Manual*.

For TNS processes on RVUs preceding the D42 RVU, *swap-file* returns the internal-format file name of the swap file for the process's data segment. This is often the name of a temporary file unless a specific swap file is supplied at run time. It can also indicate the current swap volume.

maxlen specifies the length in bytes of the string variable *swap-file*.

swap-len output

INT .EXT:ref:1

contains the actual length in bytes of the value returned in *swap-file*.

error-detail output

INT .EXT:ref:1

for some returned errors, contains additional information. See *error*.

proc-type output

INT .EXT:ref:1

returns the process type. The bits are defined as follows:

<0:14>	(reserved)	
<15>	0	Process is a Guardian process.
	1	Process is an OSS process.

oss-pid output

INT(32) .EXT:ref:1

returns the OSS process ID of an OSS process; otherwise, it returns the null OSS process ID (the null OSS process ID is obtained by calling the OSS_PID_NULL_ procedure).

General Considerations

- Process access ID (PAID) and creator access ID (CAID)

  An access ID is a word associated with a given process that contains a group ID number in the left byte and a user ID number in the right byte. Two types of access IDs are used in the operating system.

  The process access ID (PAID) is returned by PROCESS_GETINFO_ and is normally used for security checks when a process attempts to access a disk file.

  The creator access ID (CAID) is returned by PROCESS_GETINFO_ and identifies the user who created the process. It is normally used, often with the PAID, for security checks on interprocess operations such as stopping a process or creating a backup for a process.

  The PAID and the CAID usually differ only when a process is run from a program file that has the PROGID attribute set. This attribute is usually set with the File Utility Program (FUP) SECURE command and PROGID option. In such a case, the process access ID returned by PROCESS_GETINFO_ is the same as the program file's owner ID.

  For more information about access IDs, see the *Guardian User's Guide* .

- Obtaining information about a process that is terminating

  If the process specified in a call to PROCESS_GETINFO_ is in the terminating state, the procedure still returns information about that process. This differs from the behavior of some of the procedures superseded by PROCESS_GETINFO_, such as GETCRTPID and GETREMOTECRTPID, which treat a terminating process as if it did not exist.

- Return value of PROCESS_GETINFO_

  If the process specified in the call to PROCESS_GETINFO_ is in the starting or terminating stage, or if its program file or libraries are being loaded by RLD, then the procedure returns $< coldload-vol>.< coldload-subvol>.NOPROGRM as the program file name.

- Error 3 will be returned if any of the 'buffers' to accept file names are not largeenough to hold the returned filename. The current maximum size for a NSK Error-Detail for return error 2 or 3 may not be the same as the comma separatedargument list and are as follows:

  1. processhandle
  2. proc-fname or maxlen
  3. proc-fname-len
  4. priority
  5. mom's-processhandle

6. hometerm or maxlen
7. hometerm-len
8. process-time
9. creator-access-id
10. process-access-id
11. gmom's-processhandle
12. jobid
13. program-file or maxlen
14. program-len
15. swap-file or maxlen
16. swap-len
17. error-detail
18. proc-type
19. oss-pid

Mom Considerations

- Obtaining the mom (*mom's-processhandle*) of a named process or process pair

  If the specified process is a single named process (that is, the specified process is the primary process of a named process pair with no backup process), a null process handle (-1 in each word) is returned in *mom's-processhandle*.

  If the specified process is the primary process of a named process pair and there is a backup process, the process handle of the backup is returned in *mom's-processhandle*.

  If the specified process is the backup process of a named process pair, the process handle of the primary is returned in *mom's-processhandle*.

- The caller can always retrieve its own mom, if it has one.
- If another process has become the mom of the specified process by a call to PROCESS_SETINFO_ or STEPMOM, then the process handle of that other process is returned in *mom's-processhandle*.
- By default, an OSS process does not have a mom process; therefore, a null process handle is returned in *mom's-processhandle*. An OSS process can have a mom process if it was created by the OSS `tdm_fork()` or one of the `tdm_exec` set of functions; see the online reference pages or the *Open System Services System Calls Reference Manual* for details. An OSS process does have

a mom process if a mom process has been explicitly assigned by either the PROCESS_SETINFO_ or STEPMOM procedure.

Home Terminal Considerations

- The home-terminal file name returned by PROCESS_GETINFO_ is in a form suitable for passing directly to file-system procedures such as FILE_OPEN_.
- The home terminal is always the same as the home terminal of the original creator (not stepmom) of the process unless the home terminal is altered by a call to PROCESS_SETSTRINGINFO_, SETMYTERM, PROCESS_DEBUG_ or DEBUGPROCESS, or the home terminal option is supplied to PROCESS_CREATE_, PROCESS_SPAWN_, NEWPROCESS, NEWPROCESSNOWAIT, OSS `tdm_fork()`, OSS `tdm_spawn()`, or one of the OSS `tdm_exec` set of functions.

I/O Processes That Control Multiple Devices

- If *`processhandle`* is an I/O process that controls multiple devices, the returned *`proc-fname`* is the name of the first device controlled by that I/O process.

OSS Considerations

- Use this procedure to find out if a process is an OSS process and to retrieve the OSS process ID associated with the process handle.
- An OSS process can change its processor,pin value during its lifetime. Zombie processes are not returned, because processor,pin pairs are not defined for zombie processes. The OSS process ID is a unique identifier representing an OSS process. It is a positive integer. It is not reused by the system until the process lifetime ends. A zombie process is an inactive process that will be deleted by its parent process.

Example

```
error := PROCESS_GETINFO_ ( proc^handle ,
                            proc^descriptor:maxlen ,
                            proc^desc^length , ,
                            moms^proc^handle , , , , , ,
                            gmoms^proc^handle , jobid );
```

Related Programming Manual

For programming information about the PROCESS_GETINFO_ procedure, see the *Guardian Programmer's Guide*.

PROCESS_GETINFOLIST_ Procedure

Summary

The PROCESS_GETINFOLIST_ procedure obtains detailed information about a specified process or about a set of processes that meet specified criteria. You can specify processes for which information is to be returned in one of several ways:

- You can specify the process handle of a particular process.
- You can specify the node name, processor, and PIN of a particular process.
- You can specify the node name and OSS process ID of a particular OSS process.
- You can specify a node name, processor, and PIN, along with a set of search criteria; the procedure searches processes in the specified processor starting at the specified PIN. You can specify that PROCESS_GETINFOLIST_ return information for only the first process that meets the search criteria or for multiple processes that meet the search criteria.
- You can omit the first four parameters and the *oss-pid* parameter to have information returned for the calling process.

A related procedure, PROCESS_GETINFO_ , is recommended for obtaining less information about a specified process.

Syntax for C Programmers

```
#include <cextdecs(PROCESS_GETINFOLIST_)>

short PROCESS_GETINFOLIST_ ( [ short cpu ]          /* i   1 */
                      ,[ short *pin ]               /* i,o 2 */
                      ,[ char *nodename ]           /* i:i 3 */
                      ,[ short length ]             /* i:i 3 */
                      ,[ short *processhandle ]     /* i   4 */
                      ,short *ret-attr-list         /* i   5 */
                      ,short ret-attr-count         /* i   6 */
                      ,short *ret-values-list       /* o   7 */
                      ,short ret-values-maxlen      /* i   8 */
                      ,short *ret-values-len        /* o   9 */
                      ,[ short *error-detail ]      /* o  10 */
                      ,[ short srch-option ]        /* i  11 */
                      ,[ short *srch-attr-list ]    /* i  12 */
                      ,[ short srch-attr-count ]    /* i  13 */
                      ,[ short *srch-values-list ]  /* i  14 */
                      ,[ short srch-values-len ]    /* i  15 */
                    ,{ __int32_t oss-pid ] ) ;      /* i 16*/;
```

- CEXTDECS (through the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

- The parameter *length* specifies the length in bytes of the character string pointed to by *nodename*. The parameters *nodename* and *length* must either both be supplied or both be absent.

Syntax for TAL Programmers

```
error := PROCESS_GETINFOLIST_ ( [ cpu ] ! i 1 !
                                ,[ pin ] ! i,o 2 !
                               ,[ nodename: length ]     ! i:i 3!
                               ,[ processhandle ]        ! i   4!
                               , ret-attr-list           ! i   5!
                               , ret-attr-count          ! i   6!
                               , ret-values-list         ! o   7!
                               , ret-values-maxlen       ! i   8!
                               , ret-values-len          ! o   9!
                               ,[ error-detail ]         ! o  10!
                               ,[ srch-option ]          ! i  11!
                               ,[ srch-attr-list ]       ! i  12!
                               ,[ srch-attr-count ]      ! i  13!
                               ,[ srch-values-list ]     ! i  14!
                               ,[ srch-values-len ]      ! i  15!
                               ,[ oss-pid ] );           ! i 16 !
```

Parameters

error returned value

INT

indicates the outcome of the operation. It returns one of these values:

0 Information is returned for the specified process or processes; *error-detail* contains the number of processes for which information has been returned (might be more than one process if in search mode).

1 File-system error; *error-detail* contains the error number.

2 Parameter error; *error-detail* contains the number of the first parameter found to be in error, where 1 designates the first parameter on the left. Note that parameters are counted as in TAL; thus, `nodename:length` are considered together as number 3, and `processhandle` is number 4.

3 Bounds error; *error-detail* contains the number of the first parameter found to be in error, where 1 designates the first parameter on the left.

4 Specified process does not exist or does not meet search criteria. If search criteria were specified, the information returned is for a process (or processes)

with a higher PIN; *error-detail* contains the number of processes for which information has been returned (might be more than one process if in search mode). If no search criteria were specified, no information was returned and *error-detail* contains 0.

5 Unable to communicate with *cpu*; *cpu* might not exist.

6 Unable to communicate with *nodename*.

7 No more matches exist; *error-detail* contains the number of processes for which information has been returned (might be 0).

8 (reserved)

9 Invalid search attribute code; *error-detail* contains the first code in question to be detected (*error-detail* is not an index into a list).

10 Invalid search value; *error-detail* contains the associated attribute code (not an index into a list).

11 Invalid return attribute code; *error-detail* contains the code in question (*error-detail* is not an index into a list).

12 Invalid *srch-option*

14 Invalid auxiliary data size specification in an attribute code; *error-detail* contains the attribute code.

15 An iterative attribute was not the last attribute in *ret-attr-list*; *error-detail* contains the attribute code.

16 Attribute not permitted in a search request; *error-detail* contains the attribute code.

17 Attribute restricted to privileged callers; *error-detail* contains the attribute code.

cpu input

INT:value

if present and not -1, is the number of the processor of interest. *cpu* must be used with *pin*.

cpu,*pin*, and optionally *nodename*:*length*, should be specified either when a search is to be performed or when the caller is interested in a specific process but does not know its process handle. If *cpu* and *pin* are specified, *processhandle* must be omitted or null (-1 in each word) and *oss-pid* must be omitted or null (a null OSS process ID is obtained by calling the OSS_PID_NULL_ procedure).

pin input,output

INT .EXT:ref:1

if present and not -1, contains either the PIN of the process of interest or the PIN of the first process to be examined in a search. *pin* is required if *cpu* is specified.

If *srch-option* is omitted or 0, the caller is requesting information about process *pin*.

If *srch-option* is 1 or 2, the caller wants to search the processes in *cpu* for those that match a set of criteria (such as having a particular user ID and program file name). In this case, *pin* is used to indicate where to begin searching, and is usually set to 0 on the initial call. When the procedure returns, *pin* has been updated to reflect the starting point for a subsequent call. *pin* is set to -1 if no more matches would be found on a subsequent call.

nodename:*length* input:input

STRING .EXT:ref:*, INT:value

if present and *length* is not 0, specifies the name of the node on which *cpu*,*pin* or *oss-pid* resides; this parameter cannot be used with *processhandle*. If used, the value of *nodename* must be exactly *length* bytes long. If *cpu* and *pin* are specified or *oss-pid* is specified but *nodename*:*length* is omitted or *length* is 0, the local node is used.

If a remote node could be running an operating system version earlier than D30, use the applicable attributes code (73) to determine which attribute code ranges are defined for the calling process.

processhandle input

INT .EXT:ref:10

if present and not null, is an input parameter specifying the process handle of the process of interest. The null process handle is one which has -1 in each word (Refer to Guardian procedure call, PROCESSHANDLE_NULLIT_). However, PROCESS_GETINFOLIST_ also treats a process handle with -1 in the first word as a null process handle.

If a value is supplied for *processhandle*, then *srch-option* must be omitted or 0, *nodename*:*length* must be omitted or *length* must be 0, *cpu* and *pin* must be omitted or -1, and *oss-pid* must be omitted or null (a null OSS process ID is obtained by calling the OSS_PID_NULL_ procedure).

ret-attr-list input

INT .EXT:ref:*

is an array of INTs indicating the attributes, and any auxiliary data supplied with auxilary data attributes, that are to have their values returned in *ret-values-list*. See Attribute Codes and Value Representations on page 12-75 for details on the attribute format. The attributes you can specify are described in Table 12-2.

ret-attr-count input

INT:value

indicates the number of 16-bit words the caller is supplying in *ret-attr-list*. This number includes the attribute count and the word count for any auxiliary data supplied with auxiliary data attributes. Valid values for the *ret-attr-count* parameter are in the range 0 through 1024.

ret-values-list output

INT .EXT:ref:*

contains *ret-values-len* words of returned information. The values parallel the items in *ret-attr-list*. For details, see Attribute Codes and Value Representations on page 12-75. Each value begins on a word boundary. A variable-length string, such as a file name, is represented by an INT value giving the byte length of the string followed by the actual string. If a string is an odd number of bytes in length, it is followed by an unused byte whose value is indeterminate.

If *srch-option* indicates that information can be returned for multiple processes, then the *ret-values-list* might contain information for more than one process. The second process's information starts at the first word boundary following the last item of the first process's information. The procedure returns as many complete sets of values as will fit in the buffer; it does not return one or more complete sets plus a partial set.

If the return values don't fit in *ret-values-list*, the procedure returns an *error* of 1 and an *error-detail* value of 563 (buffer too small); no process information is returned.

Whenever *srch-option* is 1 or 2, the caller should either include the PIN (38) or OSS process ID (90) attribute in the set of requested attributes in order to identify the Guardian or OSS process associated with each set of returned values.

ret-values-maxlen input

INT:value

is the maximum length in words of *ret-values-list*. Valid values for the *ret-values-maxlen* parameter are in the range 0 through 8192.

ret-values-len output

INT .EXT:ref:1

is the actual length in words of *ret-values-list*.

error-detail output

INT .EXT:ref:1

for some returned errors, contains additional information. See *error*, above.

srch-option input

INT:value

has one of these values:

0 Return information for only the process specified by [*nodename*,] *cpu*,*pin* or by *processhandle*. You cannot specify *oss-pid* with this value. These parameters are ignored when *srch-option* is set to 0: *srch-attr-list*, *srch-attr-count*, *srch-values-list*, and *srch-values-len*.

1 Start a search at [*nodename*,] *cpu*,*pin* and return information for the first matching process. You cannot specify *processhandle* or *oss-pid* with this value.

2 Start a search at [*nodename*,] *cpu*,*pin* and return information for as many matching processes as will fit in *ret-values-list*. You cannot specify *processhandle* or *oss-pid* with this value.

3 Return information for only the OSS process specified by [*nodename*,] *oss-pid*. You cannot specify *processhandle* or *cpu*,*pin* with this value. These parameters are ignored when *srch-option* is set to 3: *srch-attr-list*, *srch-attr-count*, *srch-values-list*, and *srch-values-len*.

The default is 0.

If you specify a value of 1 or 2 and an *error* of 0 or 4 is returned, information has been returned for at least one process.

If you specify a value of 1 or 2 and an *error* of 7 (no more matches) is returned, the search is complete; the value of *error-detail* indicates whether any process information has been returned. If no process information has been returned, the value of *error-detail* is 0 and the value of *pin*, if specified, is -1; the values of all other output parameters are indeterminate.

srch-attr-list input

INT .EXT:ref:*

is an array of integer attribute codes specifying the items that are included in the search criteria. This parameter must be supplied if *srch-option* is 1 or 2. This parameter is ignored if *srch-option* is 0 or 3. For the list of valid codes, see Attribute Codes and Value Representations on page 12-75.

srch-attr-count input

INT:value

is the number of entries in *srch-attr-list*. This parameter must be supplied if *srch-option* is 1 or 2. This parameter is ignored if *srch-option* is 0 or 3.

srch-values-list input

INT .EXT:ref:*

is a list of the match values for the attributes in *srch-attr-list*. Its order must exactly parallel the order of the attribute codes. The value representations are described under Attribute Codes and Value Representations on page 12-75. Each value begins on a word boundary. A variable-length string, such as a file name, is represented by an INT value giving the byte length of the string followed by the actual string. If a string is an odd number of bytes in length, it is padded with an extra byte whose value is indeterminate.

This parameter must be present if *srch-option* is 1 or 2. This parameter is ignored if *srch-option* is 0 or 3.

srch-values-len input

INT:value

if present, is the length in words of *srch-values-list*.

This parameter must be present if *srch-option* is 1 or 2. This parameter is ignored if *srch-option* is 0 or 3.

oss-pid input

INT(32):value

if present and not null, contains the OSS process ID of the OSS process of interest. A null OSS process ID is obtained by calling the OSS_PID_NULL_ procedure. *nodename*:*length* should be specified when the caller wants information about an OSS process on a remote node. The attributes OSS controlling terminal (27) and OSS program pathname (93) are valid only on the local node.

If a value is supplied for *oss-pid*, then *srch-option* must be 3, *processhandle* must be omitted or null (-1 in each word), and *cpu* and *pin* must be omitted or -1. This parameter is ignored if *srch-option* is 0,1, or 2.

General Considerations

- All considerations listed under PROCESS_GETINFO_ also apply to PROCESS_GETINFOLIST_.

- You must qualify any file names used as search attributes. If the process of interest is located on a remote node, local-form file names are treated as local to that node, *not* local to the caller's node.
- All returned file names are fully qualified.
- When using PROCESS_GETINFOLIST_ procedure, if you want to get information on every process in the processor, specify a search criteria that will find every process. For example, specify Search Attribute Code 9 with 0 as the search value. Information will be returned for all processes since all processes have a priority that is greater than or equal to 0.

Auxiliary Data

Certain attributes require auxiliary data that provides additional information about the attributes. This auxiliary data is provided by the caller and immediately follows the attribute code in the *ret-attr-list* array. The word containing the attribute code includes a field in which the caller specifies the length of the auxiliary data (see Attribute Codes and Value Representations on page 12-75).

Note that if any auxiliary data is included in *ret-attr-list*, the *ret-attr-count* parameter must include the word count for the auxiliary data.

Iterative Attributes

Iterative attributes are used to report information about multiple loadfiles. These attributes return a variable length array describing object files loaded for the target process. The iterative attributes perform an iterative query, in which multiple loadfiles are queried. Each iteration is an invocation of PROCESS_GETINFOLIST_. Iteration is controlled by a 64-bit context value that is input as auxiliary data in *ret-attr-list* and returned in *ret-values-list*.

To start an iterative query, the caller initially sets the context value to zero. When the query is complete, that is, all loadfiles requested by the attribute have been reported, the returned context value is zero.

A nonzero returned context value indicates that the returned information has exceeded the amount of available space specified by *ret-values-list* and *ret-values-maxlen*. In this case, as much information as will fit in the space is returned; the caller can then copy the returned context value into the attribute's auxiliary data and call PROCESS_GETINFOLIST_ to continue iterating.

An iterative attribute must be the last one in the attribute list. If it is not, an error is reported. Iterative attributes cannot be used with *srch-option* 1 or 2.

Loadfile Types

Several attributes return information about loadfiles, including the loadfile type. The loadfile type is indicated by a code having one of the values shown in this table:

Code	ID *suffix**	Description
15	LTMASK	Mask to isolate the loadfile type; one of the values 0 through 8, as follows:
0	ERROR	Error (for example, no loadfile segment at the specified address)
1	TNSUNAXLPRG	Unaccelerated TNS program
2	TNSUNAXLUL	Unaccelerated TNS user library
3	TNSAXCLPRG	Accelerated TNS program
4	TNSAXCLUL	Accelerated TNS user library
5	NOPICELFPRG	Non-PIC ELF program
6	NOPICELFSRL	Non-PIC ELF shared run-time library (SRL)
7	PICELFPRG	PIC ELF program
8	PICELFDLL	PIC ELF dynamic-link library (DLL)
768	LIBMASK	Mask to isolate the next three values
256	PRIVATELIB	Ordinary library (not a public or implicit library)
512	PUBLICLIB	Public library (one of the set of installed public libraries)
768	IMPLIB	Implicit library
2048	OSIMAGE	This object is included in OSIMAGE
4096	DYNAMIC	This library was loaded dynamically
8192	OSSPROCESS	This file was loaded in an OSS process, so its name is in OSS format
16384	MAYSETBPT	Privilege is not required to debug (set breakpoints in) this file

* These constants are declared in DDL-based header files, including ZSYSC section process_getinfolist_return, with identifiers ZSYS_VAL_PINF_TYPE_*suffix* and ZSYSTAL section PROCESS^GETINFOLIST^RETURN with identifiers ZSYS^VAL^PINF^TYPE^*suffix*.

Using the Loadfile Type Code Values

The type code value returned is one of the values 0 through 8 plus other values as appropriate. Thus, for example, a DLL dynamically loaded into an OSS process would have a type code value of 28936, broken down as follows:

28936 = 0x7108
= 8 (DLL)

+ 256 (ordinary library)
+ 4096 (dynamically loaded)
+ 8192 (loaded in OSS process)
+ 16384 (OK to debug)

Attribute Codes and Value Representations

Each attribute code is contained in a 16-bit word defined as follows:

low-order 12 bits contains the attribute index

high-order 4 bits contains the length, in 16-bit words, of any auxiliary data

For attributes with no auxiliary data, the data length is 0.

The individual attribute codes and their associated value representations are shown in Table 12-2 on page 12-75. The attribute codes are defined symbolically in ZSYSDDL. For example, see section `process_itemcodes` in ZSYSC or `PROCESS^ITEMCODES` in ZSYSTAL. (Comments in these files see the attribute codes as "item codes".) These files are distributed in an installation subvolume named ZSYSDEFS.

Table 12-2. PROCESS_GETINFOLIST_ Attribute Codes and Value Representations (page 1 of 5)

Code	Attribute	TAL Value Representation
1*+	creator access ID	INT
2*+	process access ID	INT
3+^	maximum priority (search only)	INT
4*+	Guardian program file	INT bytelength, STRING
5*+	home terminal	INT bytelength, STRING
6*+	gmom's process handle	INT (10 words)
7*+	jobid	INT
8+	process subtype	INT
9+^	minimum priority (search only)	INT
10+	process state	INT
11+	system process type	INT (mask), INT (value)
12+^	earliest creation time (search only)	FIXED
13+^	latest creation time (search only)	FIXED
14+	lowered priority	none (as a search attribute) INT (as a return attribute)

* indicates that this attribute is also a parameter of PROCESS_GETINFO_
+ indicates that this attribute can be used as search attributes
& indicates that this attribute applies only to OSS processes
^ indicates that this attribute cannot be specified as a return attribute
@ indicates that this attribute requires auxiliary data.The data length *n* in the high-order 4 bits is shown as *n*<<12

Table 12-2. PROCESS_GETINFOLIST_ Attribute Codes and Value Representations (page 2 of 5)

Code	Attribute	TAL Value Representation
15+	process list	INT
16-20	(reserved for future use)	
21+	real group ID	INT(32)
22+	real user ID	INT(32)
23+	effective user ID	INT(32)
24, 25	(reserved for future use)	
26+&	OSS session leader	INT(32)
27+&	OSS controlling terminal	INT bytelength, STRING <= 1024
28*+	process type	INT
29	(reserved for future use)	
30*	process time	FIXED
31	wait state	INT
32	process state	INT
33+	library file	INT bytelength, STRING
34*	swap file	INT bytelength, STRING
35	context changes	INT
36	DEFINE mode	INT
37	licenses	INT
38	PIN	INT
39*	file name	INT bytelength, STRING
40*	mom's process handle	INT (10 words)
41	process file security	INT
42*	current priority	INT
43	initial priority	INT
44	remote creator	INT
45	logged-on state	INT
46	extended swap file	INT bytelength, STRING
47	primary	INT
48*	process handle	INT (10 words)

* indicates that this attribute is also a parameter of PROCESS_GETINFO_
+ indicates that this attribute can be used as search attributes
& indicates that this attribute applies only to OSS processes
^ indicates that this attribute cannot be specified as a return attribute
@ indicates that this attribute requires auxiliary data.The data length n in the high-order 4 bits is shown as n<<12

Table 12-2. PROCESS_GETINFOLIST_ Attribute Codes and Value Representations (page 3 of 5)

Code	Attribute	TAL Value Representation
49	qualifier info available	INT
50	Safeguard-authenticated logon	INT
51	force low	INT
53	creation timestamp	FIXED
54	current pages	INT
55	messages sent	INT(32)
56	messages received	INT(32)
57	receive queue length	INT
58	receive queue maximum length	INT
59	page faults	INT(32)
62	named	INT
63	stop mode	INT
64	stop request queue	INT
65	mom's file name	INT bytelength, STRING
66	gmom's file name	INT bytelength, STRING
67	Safeguard-authenticated logoff state	INT
68	inherited logon	INT
69	stop on logoff	INT
70	propagate logon	INT
71	propagate stop-on-logoff	INT
72	logon flags and states	INT
73	applicable attributes	INT
75	nice() function value	INT(32)
76	process file segment (PFS) size that is being used at a particular time	INT(32)
77	maximum PFS size used	INT(32)
80	effective group ID	INT(32)
81	saved set-group-ID	INT(32)
82	login name	INT bytelength, STRING <= 32 chars

* indicates that this attribute is also a parameter of PROCESS_GETINFO_
+ indicates that this attribute can be used as search attributes
& indicates that this attribute applies only to OSS processes
^ indicates that this attribute cannot be specified as a return attribute
@ indicates that this attribute requires auxiliary data.The data length n in the high-order 4 bits is shown as n<<12

Table 12-2. PROCESS_GETINFOLIST_ Attribute Codes and Value Representations (page 4 of 5)

Code	Attribute	TAL Value Representation
83	group list	INT n, INT(32) [0:n-1]
84	saved set-user-ID	INT(32)
90*&	OSS process ID	INT(32)
91&	OSS command	INT bytelength, STRING <= 1024 chars
92&	OSS arguments	INT bytelength, STRING <= 1024 chars
93&	OSS program pathname	INT bytelength, STRING <= 1024 chars
94&	OSS parent process ID	INT(32)
95&	OSS elapsed time	INT(64)
96&	OSS processor time	INT(64)
97&	OSS start time	INT(64)
98	OSS group leader process ID	INT(32)
99&	OSS process status	INT(32)
100	process file segment (PFS) size	INT(32)
101	server class name	INT bytelength, STRING
102	origin of main stack	INT(32)
103	current main stack size	INT(32)
104	maximum main stack size	INT(32)
105	origin of the privileged stack	INT(32)
106	current privileged stack size	INT(32)
107	maximum privileged stack size	INT(32)
108	start of global data	INT(32)
109	size of global data	INT(32)
110	start of native heap area	INT(32)
111	current size of native heap area	INT(32)
112	maximum size of native heap area	INT(32)
113	guaranteed swap space	INT(32)
115	Native shared run-time library: buffer size required for attribute 116	INT

* indicates that this attribute is also a parameter of PROCESS_GETINFO_
+ indicates that this attribute can be used as search attributes
& indicates that this attribute applies only to OSS processes
^ indicates that this attribute cannot be specified as a return attribute
@ indicates that this attribute requires auxiliary data.The data length n in the high-order 4 bits is shown as n<<12

Table 12-2. PROCESS_GETINFOLIST_ Attribute Codes and Value Representations (page 5 of 5)

Code	Attribute	TAL Value Representation
116+	Native shared run-time library file-name information (superseded by 121 and 125)	INT bytelength, STRING (as a search attribute) variable-length structure (as a return attribute)
117	TNS/R native shared run-time library: buffer size required for attribute 118	INT
118	TNS/R native shared run-time library name information (in a variable-sized array)	INT number of names, INT flags, INT name length, STRING name
119	process is TNS/R native	INT
120	(reserved for privileged use)	
121+(4<<12)@	program file and explicit library information	variable-length structure
122+(4<<12)@	dynamically loaded library information	variable-length structure
123	(reserved for future use)	
123+(4<<12)@	implicit library	variable-length structure
124+(4<<12)@	loadfile detail	variable-length structure
125^+	processes that have loaded a particular file, specified by Guardian file name	INT bytelength, STRING
126^+	processes that have loaded a particular file, specified by OSS path name	INT bytelength, STRING

* indicates that this attribute is also a parameter of PROCESS_GETINFO_
\+ indicates that this attribute can be used as search attributes
& indicates that this attribute applies only to OSS processes
^ indicates that this attribute cannot be specified as a return attribute
@ indicates that this attribute requires auxiliary data.The data length *n* in the high-order 4 bits is shown as *n*<<12

Except for attributes 125 and 126, all file names that are specified as search parameters are assumed to be sufficiently qualified and are not resolved against defaults. They are interpreted relative to the node on which the search is to be performed. For example, if a caller on node \A is inquiring about processes running on \B that have a home terminal of \A.$TERM1, then the home terminal name in the search list must be \A.$TERM1 rather than $TERM1.

File names that are in the returned values list are returned in fully qualified form.

- 1: creator access ID

See the *creator-access-id* parameter returned by the PROCESS_GETINFO_ procedure.

- 2: process access ID

  See the *process-access-id* parameter returned by the PROCESS_GETINFO_ procedure.

- 3: maximum priority

  as a search attribute, specifies the maximum priority of interest. For example, specifying a maximum priority of 199 includes all application processes in the search.

  Maximum priority cannot be specified as a return attribute code.

- 4: Guardian program file

  See the *program-file* parameter returned by the PROCESS_GETINFO_ procedure.

- 5: home terminal

  See the *hometerm* parameter returned by the PROCESS_GETINFO_ procedure.

- 6: gmom's process handle

  See the *gmom's-processhandle* parameter returned by the PROCESS_GETINFO_ procedure.

- 7: job id

  See the *jobid* parameter returned by the PROCESS_GETINFO_ procedure.

- 8: process subtype

  as a search attribute, specifies the subtype of interest. On return, it contains the subtype of the process.

  For more information about process subtypes, see General Considerations on page 12-43.

- 9: minimum priority

  as a search attribute, specifies the minimum priority of interest. Minimum priority cannot be specified as a return attribute code.

- 10: process state

  as a search attribute, specifies the process state.

  The bits are defined as follows:

  `<0:10>` (reserved)

  `<11:15>` The process state, where:

  0 unallocated process

1 starting
2 runnable (OSS process state equivalent is CONT)
3 suspended (OSS process state equivalent is STOP)
4 (reserved)
5 Debug breakpoint
6 Debug trap or signal
7 Debug request
8 Inspect memory-access breakpoint
9 Inspect breakpoint
10 Inspect trap or signal
11 Inspect request
12 saveabend
13 terminating
14 XIO initialization (not applicable on G-series RVUs)

The OSS zombie process state has no Guardian equivalent.

This attribute and the process state attribute (32) return the same information for bits <11:15>.

- 11: system process type

  as a search attribute, specifies a bit mask followed by a search value. The bit mask indicates which flags are to be searched, and the search value indicates the value of the flag to be searched. For example, to retrieve the I/O processes not configured by Dynamic System Configuration (DSC) or the Subsystem Control Facility (SCF), set bits 1 and 3 in the first word to 1, and in the second word, set bit 1 to 1 and bit 3 to 0.

  The bits are defined as follows:

<0>	System process: process is a system process.
<1>	IOP. Process is an I/O process.
<2>	(reserved)
<3>	Device is dynamically configured.
<4>	NONSTOPPROCESS: process is a privileged process that can be stopped only by either process of the process pair.
<5:15>	(reserved)

  This attribute and the process state attribute (32) return the same information for bit <0>.

- 12: earliest creation time

  as a search attribute, specifies the earliest process-creation time (in Julian timestamp format) of interest. This cannot be used as a search attribute if the target system is running an operating system version earlier than D10.

  Earliest creation time cannot be specified as a return attribute code.

- 13: latest creation time

  as a search attribute, specifies the latest process-creation time (in Julian timestamp format) of interest. This cannot be used as a search attribute if the target system is running an operating system version earlier than D10.

Latest creation time cannot be specified as a return attribute code.

- 14: lowered priority

  as a search attribute, specifies that only processes that are currently running at reduced priority due to heavy processor use are of interest. This cannot be used as a search attribute if the target system is running an operating system version earlier than D10. Note that when this attribute is included in *srch-attr-list*, there is no corresponding value in *srch-values-list*.

  When used as a return attribute, lowered priority returns 0 if the process is not currently running with its priority lowered or if the target system is running an operating system version earlier than D10; it returns 1 if the process is currently running at reduced priority due to heavy processor use.

- 15: process list

  as a search attribute, specifies the process list that a process could be on. A process cannot be on more than one process list at a time. The values are defined as follows:

  0 Process is not on any process list.
  1 (reserved)
  2 Process is on the ready list. The process is ready to run or is blocked from running by a page fault. When used as a return attribute, this option returns the same information as bit 2 of the process state attribute (32).
  3 Privileged process is on the privileged semaphore list waiting for a system semaphore.
  4 Process is on the stop list waiting to be stopped.
  5 Process is on the DMON list waiting for an Inspect request or waiting to create a saveabend file.
  6 Process is on the cleanup list waiting to clean up resources before deletion.
  7 Process is on the binary semaphore list waiting for a binary semaphore.

- 21: real group ID

  as a search attribute, specifies the real group ID.

  When used as a return attribute, this attribute returns the real group ID of the process.

  The real group ID is a process attribute that, at the time of process creation, identifies the group of the user who created the process. This value can change during the process lifetime. The group ID is a nonnegative integer that is used to identify a group of system users. Each system user is a member of at least one group.

  This attribute is undefined if the target system is running an operating system version earlier than D30.

- 22: real user ID

  as a search attribute, specifies the real user ID.

When used as a return attribute, this attribute returns the real user ID of the process.

The real user ID is a process attribute that, at the time of process creation, identifies the user who created the process. This value can change during the process lifetime. The user ID is a nonnegative integer that is used to identify a system user.

This attribute is undefined if the target system is running an operating system version earlier than D30.

- 23: effective user ID

  as a search attribute, specifies the effective user ID.

  When used as a return attribute, this attribute returns the effective user ID of the process.

  The effective user ID is a process attribute used in determining access. This value can change during the process lifetime. The user ID is a nonnegative integer that is used to identify a system user. For logged-on processes, the effective user ID is equivalent to the process access ID (PAID). For other processes, the effective user ID is invalid and there is no PAID equivalent. The effective user ID determines access to OSS disk files, and the PAID determines access to Guardian disk files.

  This attribute is undefined if the target system is running an operating system version earlier than D30.

- 26: OSS process ID of the OSS session leader (OSS processes only)

  as a search attribute, specifies the session leader.

  When used as a return attribute, this attribute returns the session leader. The session leader returned might identify a process that is no longer valid.

  The session leader is an OSS process that has created a session. A session is a collection of process groups established for job-control purposes. Each process group is a member of a session. A process is considered to be a member of the session of which its process group is a member. A newly created process joins the session of its creator. A process can alter its session membership. There can be multiple process groups in the same session.

  This attribute is undefined if the target system is running an operating system version earlier than D30.

- 27: OSS controlling terminal (OSS processes only)

  as a search attribute, specifies the controlling terminal of an OSS process as an OSS pathname. The controlling terminal is identified by a byte length followed by a string. Unlike other attributes, the string must be null-terminated and the byte length must not include the last null byte of the string. PROCESS_GETINFOLIST_ returns matching processes if the caller has the appropriate security. If the

pathname is not fully qualified, it is resolved in the current working directory (cwd) of the calling process. If it cannot be resolved, error 10 is returned.

When used as a return attribute, this attribute returns the controlling terminal of an OSS process if the controlling terminal exists. The controlling terminal is returned as a byte length followed by a string. Unlike the controlling terminal search attribute, the return attribute string is not null-terminated. A byte length of 0 is returned if either the controlling terminal does not exist, the calling process does not have the appropriate security, or OSS is not running.

Each session can have at most one controlling terminal associated with it, and a controlling terminal is associated with exactly one session.

This attribute applies only to OSS processes on the same node as the calling process.

This attribute is undefined if the target system is running an operating system version earlier than D30.

- 28: process type

  as a search attribute, if 0, specifies a search for all matching processes. If 1, this attribute specifies a search for matching OSS processes only. Note that attributes 26 and 27 search only for OSS processes regardless of the setting of this attribute.

  When used as a return attribute, this attribute returns 0 if the process is a Guardian process and 1 if the process is an OSS process. In this capacity, this attribute is equivalent to the *proc-type* parameter of the PROCESS_GETINFO_ procedure.

  This attribute is undefined if the target system is running an operating system version earlier than D30.

- 30: process time

  See the *process-time* parameter returned by the PROCESS_GETINFO_ procedure.

- 31: wait state

  returns the wait field of the process indicating what, if anything, the process is waiting on. The bits are defined as follows:

<0>	Wait on LSIG staus
<1>	Wait on LPIPE status
<2:7>	(Reserved)
<8>	Wait on PON (processor power on)
<9>	Wait on IOPON (I/O power on)
<10>	Wait on INTR (interrupt)
<11>	Wait on LINSP (Inspect event)
<12>	Wait on LCAN (message system: cancel)
<13>	Wait on LDONE (message system: done)
<14>	Wait on LTMF (TMF request)

<15> Wait on LREQ (message system: request)

The bits in the wait field are numbered from left to right. Thus, if octal 3 (%003) is returned, it means that bits 14 and 15 are equal to 1.

- 32: process state

  returns the state of the process. The bits are defined as follows:

  <0> Privileged process
  <1> Process is waiting for memory manager service, probably for a page fault.
  <2> Process is on the ready list.
  <3> System process
  <4:5> (reserved)
  <6> Memory access breakpoint in system code
  <7> Process not accepting any messages
  <8> Temporary system process
  <9> Process has logged on (called USER_AUTHENTICATE_ or VERIFYUSER).
  <10> In a pending process state
  <11:15> The process state, where:

  0 unallocated process
  1 starting
  2 runnable (OSS process state equivalent is CONT)
  3 suspended (OSS process state equivalent is STOP)
  4 (reserved)
  5 Debug breakpoint
  6 Debug trap or signal
  7 Debug request
  8 Inspect memory-access breakpoint
  9 Inspect breakpoint
  10 Inspect trap or signal
  11 Inspect request
  12 saveabend
  13 terminating
  14 XIO initialization (not applicable on G-series RVUs)

  The OSS zombie process state has no Guardian equivalent.

  This attribute and the process list attribute (15) return the same information for bit <2>. This attribute and the process state attribute (10) return the same information for bits <11:15>.

- 33: library file

  as a search attribute, specifies either a native user library file or a TNS user library file, and searches for the processes that are using it as a library.

  When used as a return attribute, the library file attribute returns the name of the library file used by the process. If the process does not have an associated library file, a length of 0 is returned.

Library file name length of 0 can also be returned if the process is in the starting or terminating stage, or, if its program file or libraries are being loaded by RLD.

- 34: swap file

  See the *swap-file* parameter returned by the PROCESS_GETINFO_ procedure.

- 35: context changes

  returns the number of changes made to the DEFINE process context since process creation, modulo 65536.

  Each process has an associated count of the changes to its context. This count is incremented each time the procedures DEFINEADD, DEFINEDELETE, DEFINESETMODE, and DEFINEDELETEALL are invoked and a consequent change to the process context occurs. In the case of DEFINEDELETE and DEFINEDELETEALL, the count is incremented by one even if more than one DEFINE is deleted. The count is also incremented if the DEFINE mode of the process is changed. If a call to CHECKDEFINE causes a DEFINE in the backup process to be altered, deleted, or added, the count for the backup process is incremented. This count is 0 for newly created processes; new processes do not inherit the count of their creators.

- 36: DEFINE mode

  returns 0 in bits <14:15> if DEFINEs are disabled; returns 1 if DEFINEs are enabled.

  Bits <0:13> are reserved and should not be assumed to contain 0.

- 37: licenses

  returns 0 in bit 15 if the program file of the process was not licensed when the process was created; returns 1 if the program file of the process was licensed when the process was created.

  Bits <0:14> are reserved and should not be assumed to contain 0.

- 38: PIN

  returns the PIN of the process whose attributes are being returned. This attribute should be specified whenever *srch-option* is not 0.

- 39: file name

See the *proc-fname* parameter returned by the PROCESS_GETINFO_ procedure.

- 40: mom's process handle

  See the *mom's-processhandle* parameter returned by the PROCESS_GETINFO_ procedure.

- 41: process file security

  returns the current default process file security setting. The security bits are as follows:

  ```
  <0:3>    0
  <4:6>    ID code allowed for read
  <7:9>    ID code allowed for write
  <10:12>  ID code allowed for execute
  <13:15>  ID code allowed for purge
  ```

  ID code can be one of these:

  ```
  0  Any user (local)
  1  Member of owner's group (local)
  2  Owner (local)
  4  Any user (local or remote)
  5  Member of owner's community (local or remote)
  6  Owner (local or remote)
  7  Super ID only (local)
  ```

- 42: current priority

  See the *priority* parameter returned by the PROCESS_GETINFO_ procedure.

- 43: initial priority

  returns the initial execution priority. If the priority has been changed by a call to PROCESS_SETINFO_ , PRIORITY, or ALTERPRIORITY, this attribute returns the new value.

- 44: remote creator

  returns 1 if the creator of the process was remote, 0 if local.

- 45: logged-on state

  returns 1 in bit <15> if the process is logged on, 0 if not.

  Bits <0:14> are reserved and should not be assumed to contain 0.

- 46: extended swap file

  returns the name of the swap file for the selectable segment that is currently in use.

  If the process is in the starting or terminating stage, or, if its program file or libraries are being loaded by RLD, then a file name length of 0 is returned.

- 47: primary

  returns 1 if the process is the current primary of a named process pair, 0 otherwise.

- 48: process handle

  returns the process handle of the process of interest.

- 49: qualifier info available

  returns 1 if the process has called PROCESS_SETINFO_ to declare that it supports qualifier name searches by the file name inquiry procedures.

  This always returns 0 if the process of interest is unnamed.

- 50: Safeguard-authenticated logon

  returns 1 if a Safeguard-authenticated logon has taken place (that is, if the process was started after successfully logging on a through terminal owned by Safeguard), 0 otherwise.

- 51: force low

  returns 1 if the process has the inherited force-low attribute set, 0 otherwise. See the description of the *create-options* parameter of PROCESS_CREATE_ for details.

- 53: creation timestamp

  returns the Julian timestamp that identifies the time when the process was created. If the target system is running an operating system version earlier than D10, 0F is returned.

- 54: current pages

  returns the number of memory pages that have been swapped in by the process and are still resident.

- 55: messages sent

  in G-series systems, returns the number of messages sent by this process since the Measure product started collecting statistics on the process. If the Measure product is not collecting statistics on the process, -1D is returned. In H-series systems, returns the number of messages sent by this process, regardless of whether or not you are using Measure.

- 56: messages received

  in G-series systems, returns the number of messages received by this process since the Measure product started collecting statistics on the process. If the Measure product is not collecting statistics on the process, -1D is returned. In H-series systems, returns the number of messages received by this process, regardless of whether or not you are using Measure.

- 57: receive queue length

  in G-series systems, returns the number of messages currently on the process receive queue. In H-series systems, returns the number of messages currently on the process receive queue, regardless of whether or not you are using Measure.

- 58: receive queue maximum length

  returns the maximum number of messages that have been on the process receive queue at any time since the Measure product started collecting statistics on the process. If the Measure product is not collecting statistics on the process, -1 is returned. This value is no longer valid in H-series systems.

- 59: page faults

  in G-series systems, returns the number of page faults for this process since the Measure product started collecting statistics on the process. If the Measure product is not collecting statistics on the process, -1D is returned. In H-series systems, returns the number of page faults for this process regardless of whether or not you are using Measure.

- 62: named

  returns 1 if the process is named, 0 otherwise.

- 63: stop mode

  returns the stop mode. For more information on the stop mode, see SETSTOP Procedure. The return values are defined as follows:

  0 Any other process can stop the process.
  1 Only qualified processes can stop the process.
  2 No other process can stop the process.

- 64: stop request queue

  returns the status of a stop request on the queue. For more information on the stop request queue, see PROCESS_STOP_ Procedure. The return values are defined as follows:

  0 A stop request is not queued.

  1 A stop request has not passed the security checks and the process is running at stop mode 1 or 2. The stop request is queued pending the reduction of the stop mode to 0.

  2 A stop request has passed the security checks but the process is running at stop mode 2. The stop request is queued pending the reduction of the stop mode to 1.

- 65: mom's file name

  returns the program file name of the mom of the process.

- 66: gmom's file name

  returns the program file name of the job ancestor of the process.

- 67: Safeguard-authenticated logoff state

  returns 1 in bit <15> if the Safeguard-authenticated logon flag is set but the process has logged off, 0 otherwise.

  Bits <0:14> are reserved and should not be assumed to contain 0.

- 68: inherited logon

  returns 1 if the logon was inherited by the process, 0 otherwise.

- 69: stop on logoff

  returns 1 if the process is to be stopped when it requests to be placed in the logged-off state, 0 otherwise.

- 70: propagate logon

  returns 1 if the process's local descendants are to be created with the inherited-logon flag set, 0 otherwise.

- 71: propagate stop-on-logoff

  returns 1 if the process's local descendants are to be created with the stop-on-logoff flag set, 0 otherwise.

- 72: logon flags and states

  returns current settings of all the logon flags and state indicators. The bits are defined as follows:

<0:8>	(reserved)
<9>	Propagate stop-on-logoff
<10>	Propagate logon
<11>	Stop on logoff
<12>	Inherited logon
<13>	Safeguard-authenticated logoff
<14>	Safeguard-authenticated logon
<15>	Logged-on state

- 73: applicable attributes

  returns the attribute types that are defined for the calling process. OSS attributes are 26, 27, and 90 through 99. Guardian extended attributes are 21 through 23, and 80 through 84. The return value of an undefined attribute is also undefined. The bits are defined as follows:

<0:13>	(reserved)
<14>	The process type, where: 0 Guardian process. Return values for OSS attributes are undefined. 1 OSS process. Return values for OSS attributes are defined.
<15>	The Guardian extended attributes, where:

0 No Guardian extended attributes. The target system is running an operating system version earlier than D30. Return values for Guardian extended attributes are undefined.
1 Guardian Extended attributes. The target system is running RVU D30 or later. Return values for Guardian extended attributes are defined.

- 76: process file segment (PFS) in use

  returns the size of the process file segment (in bytes) that is being used when the call was made.

- 77: maximum process file segment (PFS) used

  returns the maximum size of the process file segment (in bytes) that had ever been used at any time the call was made.

- 80: effective group ID

  returns the effective group ID. The effective group ID is a process attribute used in determining access. This value can change during the process lifetime. The group ID is a nonnegative integer that is used to identify a group of system users. Each system user is a member of at least one group.

  This attribute is undefined if the target system is running an operating system version earlier than D30.

- 81: saved set-group-ID

  returns the saved set-group-ID. The saved set-group-ID is a process attribute that allows some flexibility in the assignment of the effective group ID attribute.

  This attribute is undefined if the target system is running an operating system version earlier than D30.

- 82: login name

  returns the login name. The login name is either the alias name if the user was authenticated using an alias or `<group>.<user>` if the user was authenticated using a user ID. A byte length of 0 is returned if the process (or the process that created the target process) did not log in, or if the target process is created using an operating system version earlier than D30.

  This attribute is undefined if the target system is running an operating system version earlier than D30.

- 83: group list

  returns the number of groups the user belongs to followed by the group ID of each group.

  This attribute is undefined if the target system is running an operating system version earlier than D30.

- 84: saved set-user-ID

returns the saved set-user-ID. The saved set-user-ID is a process attribute that allows some flexibility in the assignment of the effective user ID attribute.

This attribute is undefined if the target system is running an operating system version earlier than D30.

- 90: OSS process ID (OSS processes only)

  See the *oss-pid* parameter returned by the PROCESS_GETINFO_ procedure.

- 91: OSS command (OSS processes only)

  returns the first 1024 bytes of the OSS command that created the process. A byte length of 0 is returned if the process is not an OSS process.

  This attribute is undefined if the target system is running an operating system version earlier than D30.

- 92: OSS arguments (OSS processes only)

  returns the first 1024 bytes of the arguments of the command that created the process. Arguments in the returned string are separated by a space. A byte length of 0 is returned if the process is not an OSS process.

  This attribute is undefined if the target system is running an operating system version earlier than D30.

- 93: OSS program pathname (OSS processes only)

  Returns the fully qualified OSS program pathname of an OSS program. This OSS attribute is the OSS equivalent of the Guardian program file attribute (4). A byte length of 0 is returned if either the calling process does not have the appropriate security, OSS is not running, or a program pathname on a remote node is requested.

  This attribute applies only to OSS processes on the same node as the calling process.

  This attribute is undefined if the target system is running an operating system version earlier than D30.

- 94: OSS parent process ID (OSS processes only)

  returns the OSS process ID of the OSS parent process; otherwise it returns the null OSS process ID (a null OSS process ID is obtained by calling the OSS_PID_NULL_ procedure). The OSS parent process ID might identify a process that is no longer active.

The OSS parent process ID is an attribute of a new process identifying the parent of the process. The parent process ID of a process is the process ID of its creator, for the lifetime of the creator.

This attribute is undefined if the target system is running an operating system version earlier than D30.

- 95: OSS elapsed time (OSS processes only)

  returns the elapsed time in microseconds since the OSS process was created; it returns 0 if the process is not an OSS process. This value is equal to the value returned by the OSS `times()` function. Note that this value is not the same as the value of the *process-time* parameter of the PROCESS_GETINFO_ procedure or the process time attribute (30) of this procedure.

  This attribute is undefined if the target system is running an operating system version earlier than D30.

- 96: OSS processor time (OSS processes only)

  returns the processor time of the OSS process ID in microseconds; it returns 0 if the process is not an OSS process. This value is equal to the system time (`tms_stime`) plus the user time (`tms_utime`) returned by the OSS `times()` function.

  This attribute is undefined if the target system is running an operating system version earlier than D30.

- 97: OSS start time (OSS processes only)

  returns the time elapsed in microseconds between the value of the OSS `TZ` environment variable and the time that the OSS process was started; it returns 0 if the process is not an OSS process. The `TZ` environment variable is usually equivalent to the system load time.

  This attribute is undefined if the target system is running an operating system version earlier than D30.

- 98: OSS process group leader process ID (OSS processes only)

  returns the OSS process ID of the OSS process group leader; otherwise, it returns the null OSS process ID (a null OSS process ID is obtained by calling the OSS_PID_NULL_ procedure). The OSS group leader process ID might identify a process that is no longer active.

  This attribute is undefined if the target system is running an operating system version earlier than D30.

- 99: process status (OSS processes only)

  returns the state of the OSS process. The bits are defined as follows:

  `<0:29>` (reserved)
  `<30>` If 1, process is a session leader.

<31> If 1, process is a group leader.

This attribute is undefined if the target system is running an operating system version earlier than D30.

- 100: process file segment (PFS) size

  returns the size of the process file segment (PFS) in bytes.

- 101: server class name

  this attribute currently does not return any valid value and should not be used.

- 102: origin of main stack

  returns the address of the origin of the main stack.

  NIL is returned if the target system is running an operating system version earlier than D40.

- 103: current main stack size

  returns the current main stack size in bytes.

  0D is returned if the target system is running an operating system version earlier than D40.

- 104: maximum main stack size

  returns the maximum size, in bytes, to which the main stack can grow.

  0D is returned if the target system is running an operating system version earlier than D40.

- 105: origin of the privileged stack

  returns the address of the origin of the privileged stack.

  NIL is returned if the target system is running an operating system version earlier than D40.

- 106: current privileged stack size

  returns the current privileged stack size in bytes.

  0D is returned if the target system is running an operating system version earlier than D40.

- 107: maximum privileged stack size

  returns the maximum privileged stack size in bytes.

  0D is returned if the target system is running an operating system version earlier than D40.

- 108: start of global data

  returns the address of the start of global data.

NIL is returned if the target system is running an operating system version earlier than D40.

- 109: size of global data

  returns the size of global data in bytes.

  0D is returned if the target system is running an operating system version earlier than D40.

- 110: start of native heap area

  returns the address of the start of the native heap area in bytes.

  NIL is returned if the target system is running an operating system version earlier than D40.

- 111: current size of native heap area

  returns the current size of the native heap area in bytes.

  0D is returned if the target system is running an operating system version earlier than D40.

- 112: maximum size of native heap area

  returns the maximum size, in bytes, to which the native heap area can grow.

  0D is returned if the target system is running an operating system version earlier than D40.

- 113: guaranteed swap space

  returns the amount of swap space reserved for use by the process in bytes.

  0D is returned if the target system is running an operating system version earlier than D40.

- 115: Native shared run-time library: buffer size required for attribute 116

  returns the size of the buffer, in bytes, for the array returned in attribute 116. 0 is returned if the process does not use native shared run-time libraries.

- 116: Native shared run-time library file-name information

returns information on native shared run-time library file names used by the process in this variable-sized array:

TAL Value Representation	Description
INT	Number of file names returned. This value indicates how many triplets of INT, INT, and STRING, as listed below, follow this value.
INT	Flag values indicate: 0 native private shared run-time library (SRL) 1 native public shared run-time library (SRL) 2 native user library shared run-time library (SRL)
INT	Length of file name.
STRING	File name. (The returned string is padded if necessary so that the next attribute returned will begin on an even-byte boundary. The padding is not counted in the reported file name length.)

Similar information about SRLs is returned, in a different format, by attribute 121, which reports all the object files loaded in the process, not just SRLs.

As a search attribute, this attribute finds processes that have loaded a particular SRL. The Guardian file name must be specified in *srch-values-list*.

Attribute 125 can perform the same search and is more efficient.

This attribute cannot be used in the same *srch-attr-list* as 125 or 126.

- 117: Native shared run-time library: buffer size required for attribute 118

  returns the size of the buffer, in bytes, for the array returned in attribute 118. 0 is returned if the process does not use native shared run-time libraries.

- 118: Native shared run-time library name information

  returns information about native shared run-time library names used by the process in this variable-sized array:

TAL Value Representation	Description
INT	Number of names returned. This value indicates how many triplets of INT, INT, and STRING, as listed below, follow this value.
INT	Flag values indicate: 0 native private shared run-time library (SRL) 1 native public shared run-time library (SRL) 2 native user library shared run-time library (SRL)
INT	Length of name.
STRING	Name. (The returned string is padded if necessary so that the next attribute returned begins on an even-byte boundary. The padding is not counted in the reported file name length.)

- 119: process is native

  returns 1 if the process is a native process; 0 otherwise.

- 121: program file and explicit library information

  returns information about the program file, SRLs, DLLs, and any user library. It does not report implicit libraries. This is an iterative attribute that requires auxiliary data: *ret-attr-list* must specify the attribute code followed by an eight-byte context value as auxiliary data. The attribute code must include the length indication, so its value is 16505 or (4<<12)+121. This attribute code and its auxiliary data must be the last elements in *ret-attr-list*. Initially, the context value must be zero; on subsequent iterations, it must be a copy of the nonzero context returned in *ret-values-list* by the previous iteration. See Auxiliary Data and Iterative Attributes on page 12-73.

  This attribute returns this information in a variable-length array:

TAL Value Representation	Description
INT	Number of loadfiles reported
INT(64)	Context value
	Loadfile information array consisting of four values for each loadfile reported (see following description)

The loadfile information array contains these entries for each loadfile reported:

TAL Value Representation	Description
INT(64)	Address of text header
INT(64)	Creation volume sequence number of loadfile
INT(32)	Logical device number of loadfile
INT(32)	Loadfile type indicator (see Loadfile Types on page 12-74)

Result value for TAL programs:

For TAL programs, the result value is an instance of `ZSYS^PINF^LOADFILE^INFO^DEF`, in which the final member (`Z^PARTIAL^INFO`) is an array occurring the number of times specified in the first member (`Z^INFONUMRET`). These structures are declared in `ZSYSTAL` section `PROCESS^GETINFOLIST^RETURN`.

Result value for C programs:

For C programs, the result value is an instance of `zsys_pinf_loadfile_info_def`, in which the final member (`z_partial_info`, of type `zsys_pinf_loadfile_partial_def_` is an array occurring the number of times specified in the first member (`z_infonumret`). These structures are declared in `ZSYSC` section `process_getinfolist_return`.

This attribute cannot be specified with a *`srch-option`* of 1 or 2.

Additional information about individual loadfiles can be obtained by using attribute 124.

- 122: dynamically loaded library information

  This attribute returns information about SRLs and DLLs that were loaded dynamically into the process. This attribute returns a subset of the information returned by attribute 121, in the same format and using the same interation paradigm. The *`ret-attr-list`* must specify the attribute code followed by an eight-byte context value as auxiliary data. The attribute code must include the length indication, so its value is 16506 or (4<<12)+122.

  This attribute cannot be specified with a *`srch-option`* of 1 or 2.

- 123: implicit library information

  On a TNS/E system, this attribute returns information about the implicit DLLs. The information is in the same format and use the same iteration paradigm as attribute index 121 (but because the number of implict DLLs is limited, iteration is seldom necessary). The *`_ret-attr-list_`* must specify the attribute code followed by an eight-byte context value as auxiliary data. The attribute code must include the length indication, so its value is 16507 or (4<<12)+123.

On a TNS/R system, this attribute returns ten bytes of zero, so the number of loadfiles reported and the context are zero. On a TNS/R system, this attribute returns ten bytes of zero, so the number of loadfiles reported and the context are zero.

The attribute cannot be specified with a *_srch-option_* of 1 or 2.

- 124: loadfile detail

  This attribute returns information about a specified loadfile. This attribute requires auxiliary data: *ret-attr-list* must specify the attribute code followed by an eight-byte context value as auxiliary data. The attribute code must include the length indication, so its value is 16508 or (4<<12)+124 (right-justified in 64 bits). To retrieve information about any loadfile in the target process, specify an address within the loadfile's text segment as auxiliary data (see Auxiliary Data on page 12-73).

  This attribute can be used to retrieve information about individual loadfiles reported by attributes 121 and 122, by specifying the reported text header address as auxiliary data.

  This information is returned:

TAL Value Representation	Description
INT(64)	Address of text header (combined text segment on native systems)
INT(64)	Creation volume sequence number of loadfile
INT(32)	Logical device number of loadfile
INT(32)	Loadfile type indicator (see Loadfile Types on page 12-74)
INT(64)	Address of text code segment (0 on native systems)
INT(64)	Address of data constant segment (0 on native systems)
INT(64)	Address of data variable segment (combined data segment on native systems; 0 if the loadfile has no data)
INT	File name length
STRING	File name (Guardian name or, for OSS processes, full path name) The returned string is padded if necessary so that the next attribute returned will begin on an even-byte boundary. The padding is not counted in the reported file name length.

  Result value for TAL programs:

  For TAL programs, the result value is an instance of `ZSYS^PINF^LOADFILE^DETAIL^DEF`. This structure is declared in `ZSYSTAL` section `PROCESS^GETINFOLIST^RETURN`.

Result value for C programs:

For C programs, the result value is an instance of `zsys_pinf_loadfile_detail_def`. This structure is declared in `ZSYSC` section `process_getinfolist_return`.

This attribute cannot be specified with a *`srch-option`* of 1 or 2.

- 125: processes that have loaded the specified Guardian loadfile

  This is a search-only attribute used to find processes that have loaded a particular loadfile. The Guardian file name must be specified in *`srch-values-list`*. If the file name is not fully qualified, the current `=_DEFAULTS DEFINE` is used to specify the omitted system, volume, and subvolume. To search for a file on another system, the resulting qualified file name and the *`nodename`* parameter must specify the same system.

  This attribute cannot be used in the same *`srch-attr-list`* as 116 or 126.

- 126: processes that have loaded the specified OSS loadfile

  This is a search-only attribute used to find processes that have loaded a particular loadfile. The OSS path name must be specified in *`srch-values-list`*. If the path name is not absolute, it is applied to the current directory (CWD). To search for a file on another system, either the specified path name or the CWD must contain the system name (for example, `/E/sierra`), and the *`nodename`* parameter must specify the same system.

  This attribute cannot be used in the same *`srch-attr-list`* as 116 or 125.

OSS Considerations

- The PROCESS_GETINFOLIST_ procedure returns as many complete sets of values as will fit in the *`ret-values-list`* buffer; it returns no partial sets. In particular, the OSS attributes OSS controlling terminal (27) and OSS program pathname (93) can return large amounts of information. The *`ret-values-list`* buffer must be large enough to accommodate all the information requested.
- To retrieve the corresponding process handle of an OSS process ID, specify the desired OSS process ID in the *`oss-pid`* parameter, specify the process handle attribute code (48) in the *`ret-attr-list`* parameter, and search for only the specified OSS process ID by setting the *`srch-option`* parameter to 3. Note that an OSS process can use a number of process handles during its lifetime.
- The OSS CONT process state is equivalent to the Guardian runnable process state. The OSS STOP process state is equivalent to the Guardian suspend process state. The OSS zombie process state has no Guardian equivalent.
- The OSS attributes OSS controlling terminal (27) and OSS program pathname (93) can be used only on the local node. For these attributes, either specify the local node in the *`nodename`*:*`length`* parameter, set *`length`* to 0, or omit the parameter.

Example

```
attr^list := 8;     ! get subdevice type only
attr^count := 1;
ret^vals^maxlen := 1;

error := PROCESS_GETINFOLIST_ ( , , , prochandle, attr^list,
                                attr^count, ret^vals^list,
                                ret^val^maxlen, ret^val^length );
```

Related Programming Manual

For programming information about the PROCESS_GETINFOLIST_ procedure, see the *Guardian Programmer's Guide*.

PROCESS_GETPAIRINFO_ Procedure

Summary

The PROCESS_GETPAIRINFO_ procedure obtains basic information about a named process or process pair. You can specify the named process or process pair that you want information about in one of several ways:

- Supply a process handle in the *processhandle* parameter. For a process pair, supply the process handle of either the primary or backup process.
- Supply a process file name in the *pair*:*maxlen* parameter.
- Perform an indexed search of the named processes on a system by supplying an initial value for the *search-index* parameter and making repeated calls. See "Considerations" for details.

To obtain additional information about a named or unnamed process, call either the PROCESS_GETINFO_ or PROCESS_GETINFOLIST_ procedure.

Syntax for C Programmers

Note. In the TNS/E environment the CEXTDECS file uses the *int* data type for 32-bit values. This is a change from the TNS and TNS/R environments where CEXTDECS uses the *long* data type for 32-bit values.

```
#include <cextdecs(PROCESS_GETPAIRINFO_)>

short PROCESS_GETPAIRINFO_ ( [ short *processhandle ] /*i 1*/
                  ,[ char *pair ]                  /* i,o:i 2*/
                  ,[ short maxlen ]                /* i,o:i 2*/
                  ,[ short *pair-length ]          /* o     3*/
                  ,[ short *primary-processhandle ]/ * o   4*/
                  ,[ short *backup-processhandle ] /* o    5*/
             ,[ __int32_t *search-index ]          /* i, o  6*/
                  ,[ short *ancst-processhandle ] /* o     7*/
                  ,[ const char *search-nodename  /* i:i   8*/
                  ,[ short length ]                /* i:i   8*/
                  ,[ short options ]               /* i,i   9*/
                  ,[ char * ancst]                 /* i,o:i10*/
                  ,[ short maxlen]                 /*i,o:i 10*/
                  ,[ short * ancst-length ]        /* o    11*/
                  ,[ short * error-detail ]);         /*o 12*/
```

- The parameter *maxlen* specifies the maximum length in bytes of the character string pointed to by *pair*, the actual length of which is returned by *pair-length*. All three of these parameters must either be supplied or be absent.
- The parameter *length* specifies the length in bytes of the character string pointed to by *search-nodename*. The parameters *search-nodename* and *length* must either both be supplied or both be absent.

Syntax for TAL Programmers

```
error := PROCESS_GETPAIRINFO_ (
          [ processhandle ]                        !i 1 !
         ,[ pair: maxlen ]                      !i,o:i 2 !
         ,[ pair-length ]                       !o     3 !
         ,[ primary-processhandle ]             !o     4 !
         ,[ backup-processhandle ]              !o     5 !
         ,[ search-index ]                      !i,o   6 !
         ,[ ancst-processhandle ]               !o     7 !
         ,[ search-nodename: length ]           !i:i   8 !
         ,[ options ]                           !i     9 !
         ,[ ancst: maxlen ]                     !i,o:i10 !
         ,[ ancst-length ]                      !o    11 !
         ,[ error-detail ] );                     !o     12 !
```

Parameters

error returned value

INT

indicates the outcome of the operation. It returns one of these values:

0 Information is returned for a process pair (not the calling process).

2 Parameter error; *error-detail* contains the number of the first parameter found to be in error, where 1 designates the leftmost parameter.

3 Bounds error; *error-detail* contains the number of the first parameter found to be in error, where 1 designates the leftmost parameter.

4 Information is returned for a single named process (can be the calling process).

5 Information is returned for a process pair where the caller is the current primary.

6 Information is returned for a process pair where the caller is the current backup.

7 No information is returned; process is unnamed (can be the calling process).

8 No information is returned; search is complete.

9 Specified process does not exist.

10 Unable to communicate with the node where the process resides.

11 Process is an I/O process, but the option to allow I/O processes was not selected.

13 Limited information is returned for a named process that is not started, but the process name is reserved.

processhandle input

INT .EXT:ref:10

if supplied, is a process handle specifying the process of interest. You can specify either the primary or backup process when seeking information about a process pair. *processhandle* is ignored if the *search-index* parameter is present.

If *processhandle* is omitted or null and *search-index* is not present:

- If *pair*:*maxlen* is present, it specifies the process or process pair of interest.
- If *pair*:*maxlen* is not present, information is returned for the caller or the process pair to which the caller belongs.

The null process handle is one which has -1 in each word (Refer to Guardian procedure call, PROCESSHANLDE_NULLIT_). However, PROCESS_GETPAIRINFO_ also treats a process handle with -1 in the first word as a null process handle.

pair:*maxlen* input, output:input

STRING .EXT:ref:*, INT:value

if present and if *maxlen* is not 0, supplies or returns the process file name of the process or process pair of interest.

The presence or absence of the parameters *processhandle* and *search-index* determine whether *pair* is an output parameter or an input parameter as follows:

If *pair* is an output parameter:

- *maxlen* specifies the length of the string variable *pair*.
- These table describes how *pair* is returned:

named process that is not started	name in the form *\node.$name* (no sequence number is returned)

If *pair* is an input parameter:

- *maxlen* specifies the length in bytes of the value supplied in *pair*.
- If *pair* is a partially qualified process name, the process name is resolved using the node name specified in the caller's =_DEFAULTS DEFINE. To resolve the process name using the caller's node name, specify bit 14 of the *options* parameter. To search for a process on a remote node, fully qualify the process name.

- If *pair* is the name of a named process that is not started, it cannot contain a sequence number.

pair-length output

INT .EXT:ref:1

if *pair*:*maxlen* is an output parameter, contains the length in bytes of the value returned in *pair*.

primary-processhandle output

INT .EXT:ref:10

returns the process handle of the primary process of a named process pair or (if the specified process is a single named process) the process handle of a single named process.

If the process is a named process that is not started, a null process handle (-1 in each word) is returned. This procedure can return information on named processes that are not running if bit 13 of the *options* parameter is set to 1.

backup-processhandle output

INT .EXT:ref:10

returns the process handle of the backup process of a named process pair.

If there is no backup process, a null process handle (-1 in each word) is returned.

search-index input, output

INT(32) .EXT:ref:1

if present and not -1D, serves as an index for searching through the named processes on a system. To use *search-index*, initialize it to 0D before issuing the first call to PROCESS_GETPAIRINFO_ ; then issue repeated calls until an *error* value of 8 (no more names) is returned. Do not alter *search-index* between calls.

See “Considerations” for details.

ancst-processhandle output

INT .EXT:ref:10

returns the process handle of the ancestor of the specified process or process pair.

If the process or process pair does not have an ancestor, a null process handle (-1 in each word) is returned.

search-nodename:*length* input:input

STRING .EXT:ref:*, INT:value

if *length* is not 0 and *search-index* is present, specifies the name of the node on which the search is to take place. PROCESS_GETPAIRINFO_ uses *search-nodename* to determine the node to search on each call, so its contents should not be altered between calls. The value of *search-nodename* must be exactly *length* bytes long and must be a valid node name.

options input

INT:value

specifies one or more options for the call as follows:

<0:12> Reserved (specify 0)

<13> 0 Return information only for running processes.

1 Also return information for named processes that are not started, but the process names are reserved.

<14> 0 Resolve a partially qualified process name in *pair* using the caller's =_DEFAULTS DEFINE.

1 Resolve a partially qualified process name in *pair* using the caller's node.

<15> 0 Return information only for named processes.

1 Also return information for I/O processes (that is, processes controlling devices or volumes). To return information on I/O processes that are not started, also set *options* .<13> to 1.

If this parameter is omitted, 0 is used.

ancst:*maxlen* input, output:input

STRING .EXT:ref:*, INT:value

if present and if *maxlen* is not 0, returns the process file name of the ancestor of the specified process or process pair. The *maxlen* parameter specifies the length in bytes of the string variable *ancst*.

ancst-length output

INT .EXT:ref:1

if *ancst* is returned, contains its actual length in bytes.

error-detail output

INT .EXT:ref:1

returns additional information about some classes of errors. See the list under *error* for details.

Considerations

- To perform an indexed search, initialize *search-index* to 0D before issuing the first call.
- Errors 11 and 12 are not returned during an indexed search. Excluded I/O processes are skipped over with no error reported.
- If PROCESS_GETPAIRINFO_ returns any value of *error* that indicates that no information is being returned, the contents of all output parameters are undefined.
- The values returned to identify the primary and backup processes reflect the current view of the operating system at the time that PROCESS_GETPAIRINFO_ was called. When the members of a named process pair voluntarily switch responsibilities, the new primary process should call the PROCESS_SETINFO_ procedure with the *primary* attribute to inform the operating system of the pair's new state.
- The *pair* parameter is the only output parameter of interest that is returned for a named process that is not started. A named process that is not started does not have any process handles (primary, backup, or ancestor) or ancestor program file name associated with it.
- When certain *error* values are returned, a null process handle (-1 in each word) or an undefined value is returned in one or more of the output process-handle parameters. The *error* values and the affected output parameters are as follows:

error	output **process-handle parameters**
2	all are undefined.
3	all are undefined.
4	*backup-processhandle* is null.
7	all are undefined.
8	all are undefined.
9	all are undefined.
10	all are undefined.
13	*primary-processhandle*, *backup-processhandle*, and *ancst-processhandle* are null.

Example

```
error := PROCESS_GETPAIRINFO_ ( prochandle, , , primary,
                                backup );
```

PROCESS_LAUNCH_ Procedure

Summary

The PROCESS_LAUNCH_ procedure creates a new process and, optionally, assigns a number of process attributes.

You can use this procedure to create only Guardian processes, although you can call it from a Guardian process or an OSS process. The program file must contain a program for execution in the Guardian environment. The program file and any user library file must reside in the Guardian name space; that is, they must not be OSS files.

You can specify that the new process be created in either a waited or nowait manner. When it is created in a waited manner, identification for the new process is returned directly to the caller. When it is created in a nowait manner, its identification is returned in a system message sent to the caller's $RECEIVE file.

DEFINEs can be propagated to a new process. The DEFINEs can come from the caller's context or from a buffer of DEFINEs saved by the DEFINESAVE procedure.

Any parameter that can specify a file name can contain a DEFINE.

Syntax for C Programmers

```
#include <cextdecs(PROCESS_LAUNCH_)>

short PROCESS_LAUNCH_ ( void *param-list            /* i   1*/
                      ,[ short *error-detail ]    /* o   2*/
                      ,[ void  *output-list ]     /* o:i 3*/
                      ,[ short maxlen ]           /* o:i 3*/
                      ,[ short *output-list-len ); /* o 4*/
```

The parameter *maxlen* specifies the maximum length in bytes of the character string pointed to by output*-list*, the actual length of which is returned by output*-list-len*. These three parameters must either all be supplied or all be absent.

Syntax for TAL Programmers

```
error:= PROCESS_LAUNCH_
            ( param-list              ! i 1 !
             ,[ error-detail ]        ! o 2 !
             ,[ output-list:maxlen ]! o:i 3 !
             ,[ output-list-len ]  );   ! o 4 !
```

Parameters

error returned value

INT

indicates the outcome of the operation. Table 12-3 on page 12-111 summarizes possible values for *error*.

param-list input

INT .EXT:ref:*

specifies the address of the ZSYS^DDL^PLAUNCH^PARMS structure that contains all of the input fields for this procedure. For information on how to assign field values to the structure, see Structure Definition for param-list on page 12-126.

error-detail output

INT .EXT:ref:*

returns additional information about some classes of errors. The sets of values for *error-detail* vary according to the *error* value, as described in Table 12-4 on page 12-120.

output*-list:maxlen* output:input

STRING .EXT:ref:*, INT:value

specifies the address of the ZSYS^DDL^SMSG^PROCCREATE structure that contains the output fields for this procedure. The ZSYS^DDL^SMSG^PROCCREATE structure is the same structure as the nowait PROCESS_LAUNCH_ and PROCESS_CREATE_ completion message. For information on field values of this structure, see "Structure Definition for output *-list*."

The value of *maxlen* determines the number of bytes of the structure that are returned to PROCESS_LAUNCH_. Note that the field, ZSYS^DDL^SMSG^PROCCREATE.PROCID cannot be truncated. If the value of *maxlen* would cause it to be truncated, fewer bytes of the structure are returned. If *maxlen* is equal to or greater than the length of ZSYS^DDL^SMSG^PROCCREATE, then the entire structure is returned.

output *-list-len* output

INT .EXT:ref:*

returns the length, in bytes, of the structure returned in output *-list*.

Table 12-3. Summary of Process Creation Errors (page 1 of 10)

error	Description
0	No error; process created, or creation initiated if you are creating the process in a nowait manner.
1	File-system error on program file; *error-detail* contains a file-system error number*.
2	Parameter error; from PROCESS_LAUNCH_ and PROCESS_SPAWN_, *error-detail* contains the literal for the first parameter to be found in error. See Table 12-4 for possible values. From PROCESS_CREATE_, *error-detail* contains the number of first parameter found to be in error, where 1 designates the leftmost parameter. Note: The PROCESS_CREATE_ parameters are counted as in TAL (see Syntax for TAL Programmers on page 12-36) rather than C. Thus, *program-file:length* is parameter 1, *library-file:length* is parameter 2, and so on; *priority* is reported as parameter 5 although it is the 9th parameter in the C calling sequence.
3	Bounds error; from PROCESS_LAUNCH_ and PROCESS_SPAWN_,*error-detail* contains the literal for the first parameter to be found in error. See Table 12-4 for possible values. From PROCESS_CREATE_,*error-detail* contains the number of first parameter found to be in error, where 1 designates the leftmost parameter. Note: The PROCESS_CREATE_ parameters are counted as in TAL (see Syntax for TAL Programmers on page 12-36) rather than C. Thus, *program-file:length* is parameter 1, *library-file:length* is parameter 2, and so on; *priority* is reported as parameter 5 although it is the 9th parameter in the C calling sequence.

* See the *Guardian Procedure Errors and Messages Manual* for a list of all file-system and DEFINE errors.

**When *error-detail* indicates the number of a shared run-time library (SRL), the number represents either the public SRL relative number, or 00 for a native user library. For more information on shared run-time libraries (SRLs) see the *nld and noft Manual.*

Note: See *Guardian Procedure Errors and Messages Manual* for Cause, Effect, and Recovery of all the Process Creation Errors.

Table 12-3. Summary of Process Creation Errors (page 2 of 10)

error	Description
4	File-system error occurred on user library file; *error-detail* contains a file-system error number*.
5	File-system error occurred on swap file; *error-detail* contains a file-system error number*.
6	File-system error occurred on extended swap file; *error-detail* contains a file-system error number*.
7	File-system error occurred while creating the process file segment (PFS); *error-detail* contains a file-system error number*.
8	Invalid home terminal (device either does not exist or is wrong device type); *error-detail* contains a file-system error number*.
9	I/O error to home terminal; *error-detail* contains a file-system error number*.
10	Unable to communicate with system-monitor process; *error-detail* contains a file-system error number*.
11	Process-name error; *error-detail* contains a file-system error number*. File-system error 44 indicates that, when trying to create a named process, either the DCT is full or there are no system-generated names available.
12	Invalid program-file format; *error-detail* subcodes are described in Table 12-5, Error Subcodes for Process Creation Errors 12, 13, 70, 76, 84, and 3xx.
13	Invalid user-library-file format; *error-detail* subcodes are described in Table 12-5, Error Subcodes for Process Creation Errors 12, 13, 70, 76, 84, and 3xx.
14	The process has undefined externals, but was started anyway (this is a warning).
15	No process control block available, or no PIN less than 255 is available.
16	Unable to allocate virtual address space.
17	Unlicensed privileged program or library.
18	Library conflict (see “General Considerations”).
19	Program file and library file specified are same file.
20	Program file has an invalid process device subtype. (See “General Considerations.”)
21	Process device subtype specified in backup process is not the same as that in the primary process.
22	Backup creation was specified, but caller is unnamed.

* See the *Guardian Procedure Errors and Messages Manual* for a list of all file-system and DEFINE errors.

**When *error-detail* indicates the number of a shared run-time library (SRL), the number represents either the public SRL relative number, or 00 for a native user library. For more information on shared run-time libraries (SRLs) see the *nld and noft Manual.*

Note: See *Guardian Procedure Errors and Messages Manual* for Cause, Effect, and Recovery of all the Process Creation Errors.

Table 12-3. Summary of Process Creation Errors (page 3 of 10)

error	Description
24	DEFINE error; *error-detail* contains either a file-system error number, a DEFINE error number*, or this error subcode.
	2 An excessive number of DEFINEs were to be propagated. See DEFINE Considerations on page 12-46.
26	Dynamic IOP error. (This error is returned only to privileged callers or to unprivileged callers attempting to use certain privileged features. On D-series RVUs, it can be returned when an I/O process is incorrectly configured. On G-series RVUs, it can be returned by an attempt to create a D-series I/O process.)
27	PFS size in program file is invalid (this error is not generated in G06.12 and later RVUs).
28	An unrecognized error number was returned from a remote system (probably running another level of software); *error-detail* contains the error number.
29	Unable to allocate a priv stack for the process.
30	Unable to lock the priv stack for the process.
31	Unable to allocate a main stack for the process.
32	Unable to lock the main stack of a native IOP. (This error is returned only to privileged callers.)
33	Security inheritance failure.
35	Internal process creation error; *error-detail* is an internal code that localizes the error.
36	Child's PFS error; *error-detail* contains a file-system error number*.
37	Unable to allocate global data for the process. If *error-detail* is non-zero, it indicates:
	1 Insufficient swap space available from KMSF
	2 Address range unavailable
	3 Process memory-segment limit exceeded
38	Unable to lock IOP global data for the process. (This error is returned only to privileged callers.)
40	The main stack maximum value, specified either by the procedure call or by the object file, is too large.
41	The heap maximum value, specified either by the procedure call or by the object file, is too large.
42	The space guarantee value, specified either by the procedure call or by the object file, is too large.

* See the *Guardian Procedure Errors and Messages Manual* for a list of all file-system and DEFINE errors.

**When *error-detail* indicates the number of a shared run-time library (SRL), the number represents either the public SRL relative number, or 00 for a native user library. For more information on shared run-time libraries (SRLs) see the *nld and noft Manual.*

Note: See *Guardian Procedure Errors and Messages Manual* for Cause, Effect, and Recovery of all the Process Creation Errors.

Table 12-3. Summary of Process Creation Errors (page 4 of 10)

error	Description
43	The process creation request specifies two files that contain the same shared run-time library (SRL) names; *error-detail* contains the numbers** of the duplicate SRLs in the form *xxyy* (where *xx* is the first SRL and *yy* is the duplicate SRL).
44	Unable to find a shared run-time library (SRL) specified by the program file; *error-detail* contains the SRL number** that could not be found.
45	Unable to find a shared run-time library (SRL) specified by another SRL; *error-detail* contains the SRL numbers** in the form *xxyy* (where *xx* is the SRL that specifies the *yy* SRL).
46	The process creation request specifies too many shared run-time libraries (SRLs); *error-detail* contains the maximum number of SRLs that can be used.
47	The program file requires fixups to a shared run-time library (SRL) but the program file is currently running; *error-detail* contains the SRL number** of the unavailable SRL.
48	A shared run-time library (SRL) requires fixups to another SRL; *error-detail* contains the SRL numbers** of the two SRLs in the form *xxyy* (where *xx* is the SRL that requires the fixup to the *yy* SRL).
49	Security violation. The program file is not licensed but a shared run-time library (SRL) containing instance data is licensed; *error-detail* contains the licensed SRL number**.
50	Security violation. Either the program file or shared run-time library (SRL) is licensed but a shared run-time library (SRL) is not licensed; *error-detail* contains the unlicensed SRL number**.
51	The program file requires a symbol from a shared run-time library (SRL) but the SRL is not exporting it; *error-detail* contains the SRL number** that does not export the required symbol. The program file specifies the shared run-time library (SRL) in its SRLINFO table.
52	The specified version, Z^VERSION, of the ZSYS^DDL^PLAUNCH^PARMS structure is not supported (PROCESS_LAUNCH_ only).
53	The specified version, Z^VERSION, of the ZSYS^DDL^PLAUNCH^PARMS structure is incompatible with the specified length, Z^LENGTH, of the structure (PROCESS_LAUNCH_ only).
54	An error occurred at an internal process creation interface.
55	The specified space guarantee, Z^SPACE^GUARANTEE (PROCESS_LAUNCH_) or Z^SPACEGUARANTEE (PROCESS_SPAWN_), cannot be allocated.

* See the *Guardian Procedure Errors and Messages Manual* for a list of all file-system and DEFINE errors.

**When *error-detail* indicates the number of a shared run-time library (SRL), the number represents either the public SRL relative number, or 00 for a native user library. For more information on shared run-time libraries (SRLs) see the *nld and noft Manual.*

Note: See *Guardian Procedure Errors and Messages Manual* for Cause, Effect, and Recovery of all the Process Creation Errors.

Table 12-3. Summary of Process Creation Errors (page 5 of 10)

error	Description
56	Internal error.
57	A shared run-time library (SRL) has undefined externals; *error-detail* contains the SRL number** that has undefined externals
58	Internal error.
59	Internal error.
60	Security violation; a shared run-time library (SRL) containing callable procedures must be licensed to be used by callable or privileged code.
61	Unable to allocate memory from system pool.
62	Mismatch between the symbolic reference in the importing module and the actual type in the exporting module.
63	There was an unresolved external reference for data.
64	Unable to honor floattype attribute. In G06.20 and later RVUs, this code is reported only for a program file; earlier RVUs use it also for librarles. *error-detail* contains one of these subcodes: 1 IEEE floating point not supported by processor 2 Unrecognized floating-point specification in file 3 Conflicting floating-point specifications (only for user library, and only in G06.19 and earlier RVUs)
65	Address references from one SRL to another require an adjustment that cannot be made because the referencing SRL is already in use.
66	Unable to honor the floattype attribute of a user library; *error-detail* contains one of these subcodes: 2 Unrecognized floating-point specification in file 4 User library specified Tandem floating-point, which mismatches the program 5 User library specified IEEE floating-point, which mismatches the program
67	Unable to honor floattype attribute of a DLL; *error-detail* contains one of these subcodes: 2 Unrecognized floating-point specification in file 4 DLL specified Tandem floating-point, which mismatches the program 5 DLL specified IEEE floating-point, which mismatches the program
68	A DEFINE named =_RLD is present but invalid; *error-detail* contains one of these subcodes:

* See the *Guardian Procedure Errors and Messages Manual* for a list of all file-system and DEFINE errors.

**When *error-detail* indicates the number of a shared run-time library (SRL), the number represents either the public SRL relative number, or 00 for a native user library. For more information on shared run-time libraries (SRLs) see the *nld and noft Manual.*

Note: See *Guardian Procedure Errors and Messages Manual* for Cause, Effect, and Recovery of all the Process Creation Errors.

Table 12-3. Summary of Process Creation Errors (page 6 of 10)

error		Description
	0	The DEFINE is not of class SEARCH
	2055	An attribute other than CLASS or SUBVOL0 is specified
	Any other value	As described for the DEFINEINFO function*.
69		A file-system error was encountered in the run-time loader (`rld`) library; *`error-detail`* contains the file-system error number.
70		Invalid file format in the runtime loader (`rld`) library; *`error-detail`* subcodes are described in Table 12-5.
71		An error occurred when loading or running the run-time loader (`rld`); *`error-detail`* contains one of these subcodes:
	9	The process abended while `rld` was running
	10	The process stopped while `rld` was running
	11	`rld` was licensed at the time the processor was loaded
	12	`rld` returned an out-of-range error value to the operating system
	16	The export digest of the file does not match the export digest of the implmp file in memory.
	19	The user attempted to use RLD as a user library. This use is not supported.
	22	RLD began processing, but did not complete the update of a loadfile.
	Any other value	An internal code indicating a problem in the construction or installation of `rld`
72		The run-time loader (`rld`) reported an internal error; *`error-detail`* is an internal code that localizes the error.
74		The process contained an unresolved reference to a function and so was not created. Contrast with error 14, which is a warning. (For DLLs and their client programs, unresolved function references are disallowed by default, but other options can be specified at link time or run time.)
75		A file-system error was encountered on a DLL; *`error-detail`* contains the file-system error number.
76		Invalid file format in a DLL; *`error-detail`* subcodes are described in Table 12-5.
77		An object file could not be loaded; *`error-detail`* contains one of these:

* See the *Guardian Procedure Errors and Messages Manual* for a list of all file-system and DEFINE errors.

**When *`error-detail`* indicates the number of a shared run-time library (SRL), the number represents either the public SRL relative number, or 00 for a native user library. For more information on shared run-time libraries (SRLs) see the *nld and noft Manual.*

Note: See *Guardian Procedure Errors and Messages Manual* for Cause, Effect, and Recovery of all the Process Creation Errors.

Table 12-3. Summary of Process Creation Errors (page 7 of 10)

error		Description
	1	A DLL requires a PIN < 255 in a process with a higher PIN.
	2	
	3	A public library requires fixed address space that is unavailable in this process.
	4	Insufficient address range is available to load the file.
	5	
	6	The process has exceeded the maximum number of memory segments.
	7	
	8	The C++ version of the specified library conflicts with one or more loadfiles loaded for this process.
	9	The loadfile is not licensed; license is required
	10	A DLL for this process is licensed or privileged and has unprotected data which requires that all loadfiles in the process be licensed. Atleast one unlicensed loadfile exists in the process.
	11	A licensed DLL or a privileged program refers to an unlicensed DLL.
	12	A process that has a licensed, but unprivileged program attempted to load an unlicensed non-public DLL.
	13	A licensed or privileged loadfile has globalized symbols.
	14	The loadfile was specified as dataResident and is not licensed, has no callable functions, and is not a program that has a priv entry point.
	15	This process can only be run by the local super ID.
	16	RLD failed to pass to the operating system a function pointer necessary to process the initialization functions, constructor callers, destructor callers, or termination functions specified to the linker.
	17	The specified loadfile was built with linker option -no_runtime_fixup, but it is not preset to load with the symbol bindings available on this system or in this process.
	18	The loadfile was built to use an Application Binary Interface version that is not supported.
78		An unsupported operation was attempted; *error-detail* contains one of these:
	1	A PIC program attempted to load an SRL other than a public SRL.
	2	A PIC program or DLL was licensed.
	3	A user library supplied for a PIC program was not a DLL.

* See the *Guardian Procedure Errors and Messages Manual* for a list of all file-system and DEFINE errors.

**When *error-detail* indicates the number of a shared run-time library (SRL), the number represents either the public SRL relative number, or 00 for a native user library. For more information on shared run-time libraries (SRLs) see the *nld and noft Manual.*

Note: See *Guardian Procedure Errors and Messages Manual* for Cause, Effect, and Recovery of all the Process Creation Errors.

Table 12-3. Summary of Process Creation Errors (page 8 of 10)

error		Description
	4	A public SRL requires another library that is not a public SRL.
	5	A public SRL is not a hybrid DLL-SRL.
	6	The specified library uses Version 1 C++, which is not supported with a PIC program.
	Values 4 and 5 imply an incorrect installation of the public SRLs.	
	7	RLD cannot be on the liblist of any file loaded when a) the program has a priv entry point, or b) the program file is licensed.
79	A resource limitation was detected by the run-time loader (`rld`); *`error-detail`* contains one of these:	
	1	The `rld` heap exceeded available KMSF space.
	2	The `rld` heap exceeded its allocated address range.
	3	The `rld` limit for handles was exceeded. (This error is reported for dynamic loading, not at process creation.)
	4	The process limit for keys was exceeded. (Keys are an operating system resource used by the loader.)
80	A failure occurred while loading or running a program that must be “dropped in” rather than run through RLD. Error details indicate that the dropped-in program is not constructed or installed correctly.	
	16	The export digest of the file does not match the export digest of the impImp file in memory.
81	A failure occurred while loading or running the TNS Emulator. Error details other than these indicate that the TNS Emulator is not constructed or installed correctly.	
	16	The export digest of the file does not match the export digest of the impImp file in memory.
82	A DEFINE is recognized by the systems, but it is not a valid DEFINE. Error details are	
	0	The define is not class SEARCH
	2055	An attribute other than CLASS or SUBVOL0 is specified
	All other details are as reported by the DEFININFO function.	
83	A file-system error occurred on the TNS Emulator while attempting process creation. The error detail contains a file-system error number.	
84	An error is detected in the file format of the TNS Emulator.	
99	.A failure occurred while attempting to preload a public DLL specified in the zreg file. Error details are:	

* See the *Guardian Procedure Errors and Messages Manual* for a list of all file-system and DEFINE errors.

**When *`error-detail`* indicates the number of a shared run-time library (SRL), the number represents either the public SRL relative number, or 00 for a native user library. For more information on shared run-time libraries (SRLs) see the *nld and noft Manual.*

Note: See *Guardian Procedure Errors and Messages Manual* for Cause, Effect, and Recovery of all the Process Creation Errors.

Table 12-3. Summary of Process Creation Errors (page 9 of 10)

error	Description
1	The export digest of the public DLL does not a match to the export digest found in the specified zreg file
2	The license value of the public DLL does not a match to the license value found in the specified zreg file.
3	The public DLL is licensed and has unprotected data.
4	The public DLL is not preset.
5	The public DLL has a priv or callable Main procedure
6	The public DLL does not support highpin.
7	The public DLL is not owned by super ID.
8	The public DLL has callable procedures and is not licensed.
9	A public DLL with this name has already been preloaded (duplicate name in zreg).
10	The text, data, or gateway of the public DLL overlaps that of another public DLL
11	The export digest attribute for this public DLL is missing from the zreg file.
104	A process cannot be created because there are insufficient resources (PROCESS_SPAWN_ only).
106	The *oss-program-file* parameter is an interpreter shell script that cannot be started (PROCESS_SPAWN_ only).
107	An error occurred during the allocation of user data space for static variables used by the system library. Z^TPCDETAIL contains the number of the file-system error that occurred (PROCESS_SPAWN_ only).
108	The calling process is not OSS. (PROCESS_SPAWN_ only).
110	The current working directory for the new process could not be obtained (PROCESS_SPAWN_ only).
111	One of the file descriptors specified to be duplicated with the OSS `dup()` function in the *fdinfo* parameter could not be duplicated. Z^TPCDETAIL contains the index into the ZSYS^DDL^FDINFO.Z^FDENTRY structure to identify which of the file descriptors failed to be duplicated. Z^ERRNO contains an OSS `dup()` function `errno` value (PROCESS_SPAWN_ only).
112	One of the file descriptors specified to be opened with the OSS `open()` function in the *fdinfo* parameter could not be opened. Z^TPCDETAIL contains the index into the ZSYS^DDL^FDINFO.Z^FDENTRY structure to identify which of the file descriptors failed to be opened. Z^ERRNO contains an OSS `open()` function `errno` value (PROCESS_SPAWN_ only).

* See the *Guardian Procedure Errors and Messages Manual* for a list of all file-system and DEFINE errors.

**When *error-detail* indicates the number of a shared run-time library (SRL), the number represents either the public SRL relative number, or 00 for a native user library. For more information on shared run-time libraries (SRLs) see the *nld and noft Manual.*

Note: See *Guardian Procedure Errors and Messages Manual* for Cause, Effect, and Recovery of all the Process Creation Errors.

Table 12-3. Summary of Process Creation Errors (page 10 of 10)

error	Description
113	The timeout value in the ZSYS^DDL^FDINFO.Z^TIMEOUT field of the *fdinfo* parameter was reached before the file descriptors specified in the *fdinfo* parameter could be opened. It is also possible that one of the file descriptors is not responding (PROCESS_SPAWN_ only).
114	A process cannot be created because privileged OSS processes are not supported (PROCESS_SPAWN_ only, before the G05 RVU.).
115	Unable to allocate global data or heap for the process (PROCESS_SPAWN_ only, before the G05 RVU).
116	Unable to propagate shared run-time library (SRL) data (PROCESS_SPAWN_ only).
3*xx*	Invalid file format on shared run-time library (SRL) number** *xx*; *error-detail* subcodes are described in Table 12-5.
4*xx*	4xx Invalid SRL DEFINE on DEFINE number** xx; error-detail could be 0 (invalidClass or ATTR) or one of the DEFINEINFO errors. For details about error values associated with DEFINEs, see the *Guardian Procedure Errors and Messages Manual.*
5*xx*	File-system error on shared run-time library (SRL) number** *xx*; *error-detail* contains a file-system error number*.
1100 through 1499	An internal error was detected within a module of the operating system; *error-detail* contains an internal code that localizes the error.
3505	Version incompatability of request between local and remote systems.

* See the *Guardian Procedure Errors and Messages Manual* for a list of all file-system and DEFINE errors.

**When *error-detail* indicates the number of a shared run-time library (SRL), the number represents either the public SRL relative number, or 00 for a native user library. For more information on shared run-time libraries (SRLs) see the *nld and noft Manual.*

Note: See *Guardian Procedure Errors and Messages Manual* for Cause, Effect, and Recovery of all the Process Creation Errors.

Table 12-4. error-detail Codes for PROCESS_LAUNCH_ and PROCESS_SPAWN_ Errors 2 and 3 (page 1 of 2)

error-detail	PROCESS_LAUNCH_ Structure or Parameter in Error	PROCESS_SPAWN_ Structure or Parameter in Error
1	Z^PROGRAM^NAME	oss-program-file
2	Z^LIBRARY^NAME	Z^LIBRARYNAME
3	Z^SWAPFILE^NAME	Z^SWAPFILENAME
4	Z^EXTSWAPFILE^NAME	Z^EXTSWAPFILENAME
5	Z^PRIORITY	Z^PRIORITY
6	Z^CPU	Z^CPU

Table 12-4. error-detail Codes for PROCESS_LAUNCH_ and PROCESS_SPAWN_ Errors 2 and 3 (page 2 of 2)

error-detail	PROCESS_LAUNCH_ Structure or Parameter in Error	PROCESS_SPAWN_ Structure or Parameter in Error
9	Z^NAME^OPTIONS	Z^NAMEOPTIONS
10	Z^PROCESS^NAME	Z^PROCESSNAME
13		
14	Z^HOMETERM^NAME	Z^HOMETERM
15	Z^MEMORY^PAGES	Z^MEMORYPAGES
16	Z^JOBID	Z^JOBID
17	Z^CREATE^OPTIONS	Z^CREATEOPTIONS
18	Z^DEFINES^NAME	Z^DEFINES
19	Z^DEBUG^OPTIONS	Z^DEBUGOPTIONS
20	Z^PFS^SIZE	Z^PFSSIZE
22	*param-list*	not returned by PROCESS_SPAWN_
23	*error-detail*	Z^TPCDETAIL
24	output-*list*	not returned by PROCESS_SPAWN_
25	output-*list-len*	not returned by PROCESS_SPAWN_
50	not returned by PROCESS_LAUNCH_	process-extension
51	not returned by PROCESS_LAUNCH_	Z^OSSOPTIONS
52	not returned by PROCESS_LAUNCH_	argv
53	not returned by PROCESS_LAUNCH_	envp
54	not returned by PROCESS_LAUNCH_	*envp* contains an invalid address.
56	not returned by PROCESS_LAUNCH_	inheritance
57	not returned by PROCESS_LAUNCH_	an internal error
58	not returned by PROCESS_LAUNCH_	fdinfo
59	not returned by PROCESS_LAUNCH_	path
60	not returned by PROCESS_LAUNCH_	inheritance-length

Table 12-5 contains descriptions of the error subcodes for errors 12, 13, 70, 76, and 3*xx*.

Table 12-5. Error Subcodes for Process Creation Errors 12, 13, 70, 76, 84, and 3*xx* (page 1 of 4)

Subcode	Meaning
1	The file is not a disk file.
2	For a program file designated in the process creation request: This is a TNS/R file in the Guardian file system and does not have file code 100 or 700, or the file is in the OSS file system and is not recognizable as a shell script or a TNS or ELF object file. This is a TNS/E file in the Guardian file system and does not have file code 100 or 800. For a file other than the one designated as the program file in the process creation request: A TNS library file was expected, but either the file is in the Guardian file system and does not have file code 100 or the file is in the OSS file system and is not recognizable as a TNS object file.
3	The file does not have the correct file structure.
4	The file requires a later RVU of the operating system.
5	Either a program lacks an entry point or an attempt was made to load a library as a program. (An entry point is specified either by a TAL or pTAL procedure having the MAIN attribute or by naming a native procedure in the `-e` linker option.)
6	Either an attempt was made to load a program as a library or a TNS user library has a MAIN procedure.
7	A TNS program file does not have data pages.
8	Either a native object file requires fixup to SRLs by the `nld` utility or a TNS object file was not prepared by the `Binder` program.
9	The file header INITSEGS is not consistent with its size.
10	The file resident size is greater than the code area length.
11	The file was not prepared by the `nld` utility or the Binder program.
12	The file has undefined data blocks.
13	The file has data blocks with unresolved references.
14	The file has too many TNS code segments.
15	Accelerated code length in the file is invalid.
16	Accelerated code address in the file is invalid.
17	Accelerated data length in the file is invalid.
18	Accelerated data address in the file is invalid.
19	The file has too many accelerated code segments.
20	The file has invalid resident areas in accelerated code.

Table 12-5. Error Subcodes for Process Creation Errors 12, 13, 70, 76, 84, and 3*xx* (page 2 of 4)

Subcode	Meaning
21	Accelerator header in the file is invalid.
22	Either UC (user code) or UL (user library) was accelerated with the wrong virtual address.
23	File has entry in native fixup list with invalid external entry-point (XEP) index value or invalid code address value.
24	Accelerated file has external procedure identifier list (EPIL), internal procedure identifier list (IPIL), or external entry-point (XEP) table with incorrect format.
25	UC (user code) or UL (user library) was accelerated using the wrong Accelerator option (UC, UL, SC, or SL).
26	The file was accelerated with an incompatible version of the Accelerator.
27	The file has an invalid callable gateway (GW) table.
28	The program file contains processor-specific code that cannot be run on the current processor.
29	Fixup of accelerated code was attempted in an object file that was not accelerated.
30	An internal structure of the file contains an error.
31	An internal structure of the file contains an error.
32	An internal structure of the file has an entry point value of 0.
33	An internal structure of the file contains an error.
34	The list of unresolved procedure names contains an error.
35	The fixup computed an invalid file offset to the code area.
36	The file has an invalid fixup item.
37	An internal structure of the file contains an error.
38	The instruction at a call site is not the type expected for its fixup item.
40	A virtual address specified in an ELF file is outside its allowed range. For example, a text or data segment is specified at an address not valid for this type of file.
42	The code area or data area is too large.
43	The file either has a gateway (GW) table but no callable procedures or has gateways that are not in the (GW) area.
44	The file codes of the program file and library file do not match. (Not generated in the G06.12 and later RVUs; see errors 5 and 6.)
45	The file being started can run only in the Guardian environment and it is being started in the OSS environment, or vice versa.
46	Either the TNS program or the TNS user library (but not both) expected the library to contain global data.

Table 12-5. Error Subcodes for Process Creation Errors 12, 13, 70, 76, 84, and 3*xx* (page 3 of 4)

Subcode	Meaning
47	Either the TNS program needs to import data from the TNS user library and the library is not exporting any data, or the library needs global data space and the program is not providing it.
48	A TNS program file uses a TNS shared run-time library (SRL) and is switching to a new library, but the program was accelerated by an old version of the Accelerator program that does not support SRL data relocation at fixup time. Use a version of the Accelerator program provided with the D30.00 or later RVU of the operating system.
49	A TNS object file has no code space.
50	The native object file is not loadable. Either it is a linkable file (such as unlinked compiler output) or it is an incorrect type of loadable file (such as a DLL encountered with a non-PIC program).
51	The program or library file does not have a valid ELF header for execution on this NonStop operating system. The file either is not targeted for this system, is not an ELF file, or has been corrupted.
52	An ELF file has a header specifying more than one instance of a segment that should be unique. The file is corrupt or was not built by a valid linker.
53	An ELF file has a header specifying more than one instance of a segment that should be unique. The file is corrupt or was not built by a valid linker.
54	The non-PIC ELF file is not loadable because it does not have a GINFO information header. An error occurred during the linking of the file, or the file is corrupt.
55	An ELF file is lacking a required segment.
56	The file specifies too many shared run-time libraries (SRLs).
57	The file specifies duplicate shared run-time libraries (SRLs).
58	The shared run-time library (SRL) does not export any procedures.
60	An ELF library file was expected, but the file either is in the Guardian file system and does not have a file code of 700, or it is in the OSS file system and is not recognizable as an ELF file.
61	Two related structures in the ELF file have inconsistent lengths.
62	An attempt was made to spawn a shell script on a remote node.
63	An inconsistency exists in the set of public SRLs.
64	Some value (other than an address) specified in an ELF file is outside its legitimate and reasonable range. The file may be corrupted.
65	The current export digest index specified in an ELF SRL file is greater than the count of export digests in that file. The file is probably corrupted.
66	The count of export digests in an ELF SRL exceeds 256. The file is probably corrupted.
67	A public SRL is marked to require a PIN < 255; this is not allowed.

Table 12-5. Error Subcodes for Process Creation Errors 12, 13, 70, 76, 84, and 3*xx* (page 4 of 4)

Subcode	Meaning
68	One of the headers that is expected to be at the front of an ELF file did not fit near enough to the front.
69	This PIC ELF file is not supported on TNS/R systems: It is licensed or it contains callable functions.
70	The ELF file is too big (EOF > 2**31 bytes).
71	A value in the TNS object file header is out of range; the file may be corrupt.
72	The EF_TANDEM_INSTANCE_DATA value in the ELF header is not consistent with the data program headers found; the file may be corrupt.
73	The p_flags in the ELF header for the resident text header are not as expected; the file may be corrupt.
74	The loadfile has resident text, but no data constant segment, and is not marked data_resident. This combination is not supported.
75	The DLL has callable functions but also has unprotected data. This is not supported.
76	An address to be stored into a relocation site does not fit in 32 bits.
77	The loadfile uses the 64-bit data model. The 64-bit data model is not supported on this system.
78	The loadfile is an import library or implicit DLL, not a program, ordinary DLL, or public DLL.

Structure Definition for *param-list*

The *param-list* parameter specifies the attributes of the new process.

In the TAL ZSYSTAL file, the structure for the *param-list* parameter is defined as:

```
STRUCT ZSYS^DDL^PLAUNCH^PARMS^DEF (*)
?IF PTAL
FIELDALIGN (SHARED2)
?ENDIF PTAL
;
   BEGIN
   INT     Z^VERSION;
   INT     Z^LENGTH;
   INT(32) Z^PROGRAM^NAME;
   INT(32) Z^PROGRAM^NAME^LEN;
   INT(32) Z^LIBRARY^NAME;
   INT(32  Z^LIBRARY^NAME^LEN;
   INT(32) Z^SWAPFILE^NAME;
   INT(32) Z^SWAPFILE^NAME^LEN;
   INT(32) Z^EXTSWAPFILE^NAME;
   INT(32) Z^EXTSWAPFILE^NAME^LEN;
   INT(32) Z^PROCESS^NAME;
   INT(32) Z^PROCESS^NAME^LEN;
   INT(32) Z^HOMETERM^NAME;
   INT(32) Z^HOMETERM^NAME^LEN;
   INT(32) Z^DEFINES^NAME;
   INT(32) Z^DEFINES^NAME^LEN;
   INT(32) Z^NOWAIT^TAG;
   INT(32) Z^PFS^SIZE;
   INT(32) Z^MAINSTACK^MAX;
   INT(32) Z^HEAP^MAX;
   INT(32) Z^SPACE^GUARANTEE;
   INT(32) Z^CREATE^OPTIONS;
   INT     Z^NAME^OPTIONS;
   INT     Z^DEBUG^OPTIONS;
   INT     Z^PRIORITY;
   INT     Z^CPU;
   INT     Z^MEMORY^PAGES;
   INT     Z^JOBID;
   INT     END_NOV95[0:-1];!used only to determine length
   END;
```

For TAL programs, these default values, defined in the P_L_DEFAULT_PARMS_ define in the SYSTEM.DLAUNCH file, must be specified when an option is not wanted:

Field Name	Default Value
Z^VERSION	1
Z^LENGTH	$OFFSET(PROCESS_LAUNCH_PARMS_.END_NOV95)
Z^PROGRAM^NAME	%HFFFC0000%D
Z^PROGRAM^NAME^LEN	0D
Z^LIBRARY^NAME	%HFFFC0000%D
Z^LIBRARY^NAME^LEN	0D
Z^SWAPFILE^NAME	%HFFFC0000%D
Z^SWAPFILE^NAME^LEN	0D
Z^EXTSWAPFILE^NAME	%HFFFC0000%D
Z^EXTSWAPFILE^NAME^LEN	0D
Z^PROCESS^NAME	%HFFFC0000%D
Z^PROCESS^NAME^LEN	0D
Z^HOMETERM^NAME	%HFFFC0000%D
Z^HOMETERM^NAME^LEN	0D
Z^DEFINES^NAME	%HFFFC0000%D
Z^DEFINES^NAME^LEN	0D
Z^NOWAIT^TAG	-1D
Z^PFS^SIZE	0D
Z^MAINSTACK^MAX	0D
Z^HEAP^MAX	0D
Z^SPACE^GUARANTEE	0D
Z^CREATE^OPTIONS	0D
Z^NAME^OPTIONS	0
Z^DEBUG^OPTIONS	%100000
Z^PRIORITY	-1
Z^CPU	-1
Z^MEMORY^PAGES	0
Z^JOBID	-1

In the C `zsysc` file, the structure for the *`param-list`* parameter is defined as:

```
#pragma fieldalign shared2 __zsys_ddl_plaunch_parms
typedef struct __zsys_ddl_plaunch_parms
{
   short                            z_version;
   short                            z_length;
   zsys_ddl_char_extaddr_def        z_program_name;
   long                             z_program_name_len;
   zsys_ddl_char_extaddr_def        z_library_name;
   long                             z_library_name_len;
   zsys_ddl_char_extaddr_def        z_swapfile_name;
   long                             z_swapfile_name_len;
   zsys_ddl_char_extaddr_def        z_extswapfile_name;
   long                             z_extswapfile_name_len;
   zsys_ddl_char_extaddr_def        z_process_name;
   long                             z_process_name_len;
   zsys_ddl_char_extaddr_def        z_hometerm_name;
   long                             z_hometerm_name_len;
   zsys_ddl_char_extaddr_def        z_defines_name;
   long                             z_defines_name_len;
   long                             z_nowait_tag;
   long                             z_pfs_size;
   long                             z_mainstack_max;
   long                             z_heap_max;
   long                             z_space_guarantee;
   long                             z_create_options;
   short                            z_name_options;
   short                            z_debug_options;
   short                            z_priority;
   short                            z_cpu;
   short                            z_memory_pages;
   short                            z_jobid;
} zsys_ddl_plaunch_parms_def;
```

Note that in the C `zsysc` file, the type `zsys_ddl_char_extaddr_def` is defined as long. The type char_far* is the equivalent to the type `zsys_ddl_char_extaddr_def` in DLAUNCHH. Therefore, do not use the structure definition from `zsysc` and the default structure value from DLAUNCHH.

C programs should initialize the P_L_DEFAULT_PARMS_ define in the $SYSTEM.SYSTEM.DLAUNCHH header file.

Z^VERSION

identifies the version of the ZSYS^DDL^PLAUNCH^PARMS structure.

This table summarizes the possible values for Z^VERSION. TAL literals are defined in the ZSYSTAL file. Literals in the zsysc file, for C programs, are the same as those for TAL except that they contain the underscore (_) character instead of the circumflex (^) character.

This value must be supplied:

Name (ZSYS^VAL^)	Value	Description
PLAUNCH^PARMS^VER	1	The current version of the structure

Z^LENGTH

is the length of the ZSYS^DDL^PLAUNCH^PARMS structure. Because the structure is subject to change, Z^LENGTH is used by PROCESS_LAUNCH_ to further identify the version of the structure.

Z^PROGRAM^NAME

if Z^PROGRAM^NAME^LEN is not 0, specifies the address of a string containing the name of the program file to be run. If used, the value of Z^PROGRAM^NAME must point to a valid file name and must be exactly Z^PROGRAM^NAME^LEN bytes long. The file must reside in the Guardian name space and must contain a program for execution in the Guardian environment.

The new process is created on the node where the program file resides. If the program file name is partially qualified, it is resolved using the =_DEFAULTS DEFINE. If you specify a file on the subvolume $SYSTEM.SYSTEM and the file is not found, PROCESS_LAUNCH_ then searches on the subvolume $SYSTEM.SYS*nn*.

For a description of file-name syntax, see Appendix D, File Names and Process Identifiers.

This parameter must be supplied unless the caller is creating its backup process.

Z^PROGRAM^NAME^LEN

specifies the length, in bytes, of the Z^PROGRAM^NAME field.

Z^LIBRARY^NAME

if specified and if Z^LIBRARY^NAME^LEN is not 0 or -1, specifies the address of a string containing the name of the user library file to be used by the process. If used, the string must be exactly Z^LIBRARY^NAME^LEN bytes long. If the library file name is partially qualified, it is resolved using the =_DEFAULTS DEFINE. The user library file must be on the same node as the process being created and must reside in the Guardian name space. For the program to create a linkage to the library file, the caller must have write permission to the program file.

If Z^LIBRARY^NAME is specified, unresolved external references are resolved first from the specified Z^LIBRARY^NAME, then from the system library.

If Z^LIBRARY^NAME is specified and Z^LIBRARY^NAME^LEN is -1, then the linkage to the library file used by the process when it was last run is removed, and the process runs with no library file. (The references that were previously resolved from the user library are resolved from the system library.) For the program to remove a linkage to a library file, the caller must have write permission to the program file.

If the nil pointer is provided or if Z^LIBRARY^NAME^LEN is 0, then the program runs with the same library file as it did the last time it was run (or with no file if that was how it was run) or with the library file currently executing. Write permission to the program file is not required. For more information about TNS user libraries, see the *Binder Manual*. For more information about TNS/R native user libraries and shared run-time libraries, see the *nld and noft Manual*. For more information about dynamic-link libraries (including native user libraries used with PIC programs), see the *ld and rld Reference Manual*.

If an external reference cannot be resolved, it is modified to invoke the debugger when referenced. PROCESS_LAUNCH_ then returns a warning 14 and issues a warning message to the home terminal the first time the program is run. (The warning 14 and the terminal message are issued again the first time the program is run following a system load.)

`Z^LIBRARY^NAME^LEN`

specifies the length, in bytes, of the Z^LIBRARY^NAME field.

`Z^SWAPFILE^NAME`

is not used, but you can provide it for informational purposes. If supplied, the swap file must be on the same system as the process being created. If the supplied name is in local form, the system where the process is created is assumed. Processes swap to a file that is managed by the Kernel-Managed Swap Facility. For more information on this facility, see the *Kernel-Managed Swap Facility (KMSF) Manual*. To reserve swap space for the process, specify the Z^SPACE^GUARANTEE field. Alternatively, use the `nld` utility to set native process attributes.

For TNS processes on RVUs preceding the D42 RVU, if supplied and if `Z^SWAPFILE^NAME^LEN` is not 0, this parameter specifies the address of a string containing the name of a file to be used as the swap file for the user data stack segment of the process. If used, the string must be exactly `Z^SWAPFILE^NAME^LEN` bytes long. If the swap file name is partially qualified, it is resolved using the =_DEFAULTS DEFINE. The swap file must be on the same node as the process being created and must be an unstructured file.

See “Considerations” for more information about swap files.

Z^SWAPFILE^NAME^LEN

specifies the length, in bytes, of the Z^SWAPFILE^NAME field.

Z^EXTSWAPFILE^NAME

for TNS processes, if not specified or Z^EXTSWAPFILE^NAME^LEN is 0, the Kernel-Managed Swap Facility (KMSF) allocates swap space for the default extended data segment of the process. For more information on this facility, see the *Kernel-Managed Swap Facility (KMSF) Manual*.

For TNS processes, if specified and if Z^EXTSWAPFILE^NAME^LEN is not 0, this parameter specifies the address of a string containing the name of a file to be used as the swap file for the default extended data segment of the process. If used, the string must be exactly Z^EXTSWAPFILE^NAME^LEN bytes long. If the swap file name is partially qualified, it is resolved using the =_DEFAULTS DEFINE. The swap file must be on the same node as the process being created and must be an unstructured file.

For native processes, this parameter is ignored, because native processes do not need an extended swap file.

See “Considerations” for more information about swap files.

Z^EXTSWAPFILE^NAME^LEN

specifies the length, in bytes, of the Z^EXTSWAPFILE^NAME field.

Z^PROCESS^NAME

if Z^NAME^OPTIONS is 1 and Z^PROCESS^NAME^LEN is not 0, specifies the address of a string containing the name to be assigned to the new process. If used, the string must be exactly Z^PROCESS^NAME^LEN bytes long. The name can include a node name, but the node must match that of the program file. For information about reserved process names, see General Considerations on page 12-140, and Appendix B, Reserved Process Names.

For other values of Z^NAME^OPTIONS, set Z^PROCESS^NAME^LEN to 0.

Z^PROCESS^NAME^LEN

specifies the length, in bytes, of the Z^PROCESS^NAME field.

Z^HOMETERM^NAME

if supplied and if Z^HOMETERM^NAME^LEN is not 0, specifies the address of a string containing the file name that designates the home terminal for the new process. If used, the string must be exactly Z^HOMETERM^NAME^LEN bytes long. If Z^HOMETERM^NAME is partially qualified, it is resolved using the =_DEFAULTS DEFINE.

Z^HOMETERM^NAME can be a named or unnamed process. The default value is the home terminal of the caller.

Z^HOMETERM^NAME^LEN

specifies the length, in bytes, of the Z^HOMETERM^NAME field.

Z^DEFINES^NAME

if supplied and if Z^DEFINES^NAME^LEN is not 0, specifies the address of a string containing a set of DEFINEs to be propagated to the new process. The string must be exactly Z^DEFINES^NAME^LEN bytes long. The set of DEFINEs should have been created through one or more calls to the DEFINESAVE procedure. For all cases except backup creation, DEFINEs are propagated according to the values specified in Z^CREATE^OPTIONS. For details, see DEFINE Considerations on page 12-46.

When a process creates its backup, all the caller's DEFINEs are propagated regardless of Z^CREATE^OPTIONS. If Z^DEFINES^NAME is specified, it is ignored.

Z^DEFINES^NAME^LEN

specifies the length, in bytes, of the Z^DEFINES^NAME field.

Z^NOWAIT^TAG

if specified and not -1D, indicates that the process is to be created in a nowait manner; the procedure returns as soon as process creation is initiated. For details, see Nowait Considerations on page 12-46.

If Z^NOWAIT^TAG is -1D, the process is created in a waited manner.

Z^PFS^SIZE

meaningful only if the process is being created on a pre-G06 RVU. On G06 and later RVUs, this value is range checked, but is otherwise ignored.

If present and nonzero, this parameter specifies the size in bytes of the process file segment (PFS) of the new process. In G-series RVUs, maximum PFS size is 8 MB. In H-series RVUs, maximum PFS size is 32 MB. A value in this range overrides the `nld` or Binder value stored in the program file. If you omit *`pfs-size`* or specify 0:

- the `nld` or Binder value is used if it is nonzero
- a default value is used otherwise

Z^MAINSTACK^MAX

specifies the maximum size, in bytes, of the process main stack. The specified size cannot exceed 32 megabytes (MB).

The default value of 0D indicates that the main stack can grow to 1 MB in the TNS/R enviroment and to 2 MB in the TNS/E environment. For most processes, the default value is adequate.

Z^HEAP^MAX

for native processes only, specifies the maximum size, in bytes, of the process heap. Note that the sum of the size of the heap and the size of global data cannot exceed 1.1 gigabytes (GB).

The default value of 0D indicates that the heap can grow to the default value of 1.1 gigabytes (GB) less the size of the globals. The initial heap size of a process is zero bytes. For most processes, the default value is adequate.

It is recommended that the value of *Z^HEAP^MAX* parameter should be set to zero. The developer then sets an appropriate value for *Z^HEAP^MAX* in the object file of the application depending on the kind of application, the maximum memory required and the system configuration. *Z^HEAP^MAX* then defaults to the value stored in the object file of the application to be launched.

An outline of existing limitations:

Native C and C++ programs can have up to 1.1 GB of heap. CISC objects can have up to 127.5 megabytes (MB) of heap. However, other demands for memory space can deplete the amount of memory available for heap.

Z^SPACE^GUARANTEE

specifies the minimum size, in bytes, of the amount of space that the process reserves with the Kernel-Managed Swap Facility for swapping. For more information on this facility, see the *Kernel-Managed Swap Facility (KMSF) Manual*. The value provided is rounded up to a page size boundary of the processor. If the requested amount of space is not available, PROCESS_LAUNCH_ returns error 55.

When the default value of 0D is used, the amount of space reserved is determined by the value specified in the object file for a native process or by the operating system for a TNS or accelerated process.

Z^CREATE^OPTIONS

provides information about the environment of the new process.

This table summarizes the possible values for Z^CREATE^OPTIONS. TAL literals are defined in the ZSYSTAL file. Literals in the zsysc file, for C programs, are the same as those for TAL except that they contain the underscore (_) character instead of the circumflex (^) character.

Valid values for Z^CREATE^OPTIONS are one or more of these:

Name (ZSYS^VAL^)	Value	Description
PCreatOpt^AllDefines	16	Propagate DEFINEs in Z^DEFINES and DEFINEs in the caller's context. In case of name conflicts, use the ones in Z^DEFINES. Otherwise, propagate DEFINEs as specified by other values.
PCreatOpt^AnyAncestor	64	If the caller is named, the process deletion message, if any, will go to whatever process has the calling process's name (regardless of sequence number) at that time.
PCreatOpt^Default	0	The default value, which is described with each of the other options.
PCreatOpt^DefEnabled	2	See PCreatOpt^DefOverride.
PCreatOpt^DefineList	8	Propagate DEFINEs in Z^DEFINES only. Otherwise, propagate only the DEFINEs in the caller's context.
PCreatOpt^DefOverride	4	Enable DEFINEs if PCreatOpt^DefEnabled is specified. Disable DEFINEs if PCreatOpt^DefEnabled is not specified. Otherwise, use caller's DEFINE mode.
PCreatOpt^FrcLowOver	32	Ignore the value of the caller's inherited force-low PIN attribute. Otherwise, use the value of the caller's inherited force-low PIN attribute.
PCreatOpt^LowPin	1	Require low PIN (in range 0 through 254). Otherwise, assign any PIN.

If you specify ZSYS^VAL^PCREATOPT^LOWPIN, the program is run at a low PIN. If you do not specify ZSYS^VAL^PCREATOPT^LOWPIN, the program runs at a PIN of 256 or higher if its program file and library file (if any) have the HIGHPIN program-file flag set and if a high PIN is available. However, if the calling process has the inherited force-low attribute set, the new process is forced into a low PIN even if all the other conditions for running at a high PIN are met. For further information on compatibility, see the *Guardian Programmer's Guide* and the *Guardian Application Conversion Guide*. See also DEFINE Considerations for more information on DEFINEs.

Z^NAME^OPTIONS

specifies whether the process is to be named and, if so, whether the caller is supplying the name or the system must generate it.

This table summarizes the possible values for Z^NAME^OPTIONS. TAL literals are defined in the ZSYSTAL file. Literals in the zsysc file, for C programs, are the same as those for TAL except that they contain the underscore (_) character instead of the circumflex (^) character.

One of these values must be supplied:

Name (ZSYS^VAL^)	Value	Description
PCreatOpt^CallersName	3	Process is named; name is the same as that of the caller. This option is used only for the creation of the caller's backup process
PCreatOpt^NamedBySys	2	Process is named; the system must generate a name. The generated name is four characters long.
PCreatOpt^NamedBySys 5	4	Process is named; the system must generate a name. The generated name is five characters long.
PCreatOpt^NameInCall	1	Process is named; name is supplied in Z^PROCESS^NAME.
PCreatOpt^NoName	0	Process is not named; it can be named if the RUNNAMED program-file flag is set.

If either the program file or the library file (if any) has the RUNNAMED program-file flag set, the system generates a name. The generated name is four characters long, unless Z^NAME^OPTIONS is ZSYS^VAL^PCREATOPT^NAMEDBYSYS5. In which case, the name is five characters long.

To create a backup process, set Z^NAME^OPTIONS to 3, Z^PROCESS^NAME^LEN to 0, and Z^PROGRAM^NAME^LEN to 0.

Z^DEBUG^OPTIONS

sets the debugging attributes for the new process.

This table summarizes the possible values for Z^DEBUG^OPTIONS. TAL literals are defined in the ZSYSTAL file. Literals in the zsysc file, for C programs, are the same as those for TAL except that they contain the underscore (_) character instead of the circumflex (^) character.

Valid values for Z^DEBUG^OPTIONS are as follows:

Name (ZSYS^VAL^)	Value	Description
PCreatOpt^DbgOverride	2	Use the debugger and saveabend options specified regardless of program-file flag settings. Otherwise, use the program-file flag settings. Debugger and saveabend options are specified by PCreatOpt^SaveAbend and PCreatOpt^RUND described in this table.
PCreatOpt^Default	0	The debugger and saveabend default values are set from the flags in the program file (set either by compiler directives at compile time, nld flag at link time, or Binder command at bind time) after these options are ORed with the corresponding states of the calling process.
PCreatOpt^INSPECT	1	Use the Inspect debugger. Otherwise, use the debugger specified by the program-file flag settings.
PCreatOpt^RUND	8	Enter Debug or the Inspect debugger at the first executable instruction of the program's MAIN procedure. If this option is not selected, begin normal program execution.
PCreatOpt^SaveAbend	4	If the process traps, create a saveabend file and use the Inspect debugger (regardless of whether PCreatOpt^INSPECT is selected). If this option is not selected and the process traps, do not create a saveabend file.

`Z^PRIORITY`

is the initial execution priority to be assigned to the new process. Execution priority is a value in the range 1 through 199, where 199 is the highest possible priority.

If you specify the default value of -1, the priority of the caller is used. If you specify either 0 or a value greater than 199, error 2 is returned.

Z^CPU

specifies the processor in which the new process is to run. If you specify -1, the processor is chosen as follows:

Backup process:	determined by system
Other process on local system:	same processor as caller
Process on remote system:	determined by system

The processor number of the new process can be obtained by passing the ZSYS^DDL^SMSG^PROCCREATE.Z^PHANDLE field of the output *-list* parameter to the PROCESSHANDLE_DECOMPOSE_ procedure.

Z^MEMORY^PAGES

For TNS processes, specifies the minimum number of 2048-byte memory pages allotted to the new process for user data. The actual amount of memory allocated is processor-dependent. If Z^MEMORY^PAGES is either omitted or less than the value previously assigned either by a compiler directive at compile time or by a Binder command at bind time, the previously assigned value is used. In any case, the maximum number of pages permitted is 64.

For native processes, this parameter is ignored. To specify the maximum size of the main stack, specify the `Z^MAINSTACK^MAX` field. Alternatively, use the `nld` utility to set the TNS/R process attributes or the `eld` utility to set the TNS/E process attributes.

Z^JOBID

is an integer (job ID) that specifies the job to be created. The new process is the first process of the job, and the caller is the job ancestor of the new process. This value is used by the NetBatch scheduler. For information about how to use this parameter, see Batch Processing Considerations on page 12-47.

The default value of -1 indicates that the new process is a member of the same batch job as the creator. If the creator is not part of a batch job, then neither is the new process.

Structure Definition for *output-list*

The output-*list* parameter provides information on the outcome of the PROCESS_LAUNCH_ procedure call. The structure returned is the same structure as the nowait PROCESS_LAUNCH_ and PROCESS_CREATE_ completion message.

In the TAL ZSYSTAL file, the structure for the output-*list* parameter is defined as:

```
STRUCT       ZSYS^DDL^SMSG^PROCCREATE^DEF (*)
?IF PTAL
FIELDALIGN (SHARED2)
?ENDIF PTAL
;
   BEGIN
   INT          Z^MSGNUMBER;
   INT(32)      Z^TAG;
   STRUCT       Z^PHANDLE;
      BEGIN
      STRUCT       Z^DATA;
         BEGIN
         STRING       ZTYPE;
         FILLER       19;
         END;
      INT          Z^WORD[0:9] = Z^DATA;
      STRUCT       Z^BYTE = Z^DATA;
         BEGIN STRING BYTE [0:19]; END;
      END;
   INT          Z^ERROR;
   INT          Z^ERROR^DETAIL;
   INT          Z^PROCNAME^LEN;
   INT          Z^RESERVED[0:3];
   STRUCT       Z^DATA;
      BEGIN
      FILLER       50;
      END;
   STRUCT       Z^PROCNAME = Z^DATA;
      BEGIN STRING BYTE [0:49]; END;
   END;
```

In the C zsysc file, the structure for the output *-list* parameter is defined as:

```
typedef struct __zsys_ddl_smsg_proccreate
{
   short                 z_msgnumber;
   long                  z_tag;
   zsys_ddl_phandle_def z_phandle;
   short                 z_error;
   short                 z_error_detail;
   short                 z_procname_len;
   short                 z_reserved[4];
   union
   {
      struct
      {
         signed char     filler_0[50];
      } z_data;
      char               z_procname[50];
   } u_z_data;
} zsys_ddl_smsg_proccreate_def;
```

Z^MSGNUMBER

if it contains the returned value of -102, indicates process creation.

Z^TAG

if it contains the returned value of -1D, indicates a waited request. Other values indicate the value specified in the ZSYS^DDL^PLAUNCH^PARMS.Z^NOWAIT^TAG of the *param-list* parameter.

Z^PHANDLE

for a waited request, returns the process handle of the new process. If an error occurs (Z^ERROR or Z^ERROR^DETAIL is not 0), then the returned value is -1D.

If you created the process in a nowait manner, then the returned value is the null process handle. You can retrieve the process handle from the completion message sent to $RECEIVE.

Z^ERROR

indicates the outcome of the operation. Z^ERROR is the same value as the returned value in *error*. Table 12-4 on page 12-120 summarizes the possible values for Z^ERROR.

Z^ERROR^DETAIL

returns additional information about some classes of errors. Z^ERROR^DETAIL is the same value as the *error-detail* parameter.

Z^PROCNAME^LEN

for a waited request, returns the length in bytes of the process descriptor of the new process.

If you created the process in a nowait manner, then the returned value is 0. You can retrieve the process descriptor length in the completion message sent to $RECEIVE.

Z^PROCNAME

for a waited request, returns the process descriptor of the new process.

If you created the process in a nowait manner, then the returned value is an array of zeroes. You can retrieve the process descriptor in the completion message sent to $RECEIVE.

General Considerations

- Partially qualified file names are resolved using the contents of the caller's =_DEFAULTS DEFINE. If a node name is not present in either the file name or the appropriate attribute of the DEFINE, the resolved name will include the caller's node.

  See below for details on resolution of specific file-name parameters.

- For TNS and accelerated processes on RVUs preceding the D42 RVU, if Z^SWAPFILE^NAME or Z^EXTSWAPFILE^NAME :

  - is specified and a file with that name exists, that file is used for memory swaps of the user data stack (swap file) or the default extended data segment (extended swap file) during execution of the process; if no file of that name exists, then a file of that name and of the necessary size is created and used for swaps. If the file name is partially qualified, the system uses the =_DEFAULTS DEFINE to resolve it.
  - specifies the name of a temporary file that is already in use, an error is returned.
  - specifies only the disk volume name, then a temporary file is created on the specified disk device.
  - is not specified or Z^SWAPFILE^NAME^LEN is 0, then the SWAP volume name in the =_DEFAULTS DEFINE is used if available. Otherwise, the system chooses where to place the file.

- Creation of the backup of a named process pair

  If the backup of a named process pair is created, the backup process becomes the "creator" or mom of the primary (that is, of the caller to PROCESS_LAUNCH_) and the primary becomes the mom of the newly created backup process. See the discussions of "mom process" and "ancestor process" in the *Guardian Programmer's Guide*.

- Program file and user library file differences

  A user library is an program file containing one or more procedures. The difference between a program file and a library file is that the library file cannot contain a MAIN procedure; a program file must contain a MAIN procedure. Undefined external references in a program file are resolved from the user library, if any, or the system library. Unresolved references in a library are resolved only from the system library.

- Library conflict: PROCESS_LAUNCH_ error

  The library file for a process can be shared by any number of processes. However, when a program is shared by two or more processes, all non-PIC processes must have the same user library configuration; that is, all non-PIC processes sharing the program either have the same user library, or they have no user library. A library conflict error occurs when there is already a copy of the non-PIC program running with a library configuration different from that specified in the call to PROCESS_LAUNCH_.

  This error is also generated if a user library file is specified when running an older TNS program containing an implicit user library. (Before the D30.00 RVU, a large TNS program file could be created with 16 segments of user code and up to 16 additional segments mapped as a user library. Subsequently, the user code and user library limits were raised to 32 segments each, and the binder stopped creating programs with an implicit user library.)

- Device subtypes for named processes (process subtypes)

  The device subtype (or process subtype) is a program file attribute that can be set by either a TAL compiler directive at compile time, `nld` flag at link time, or Binder command at bind time. You can obtain the device type and subtype of a named process by calling FILE_GETINFO[BYNAME]_ , FILEINFO, or DEVICEINFO.

  Note that a process with a device subtype other than 0 must always be named.

  There are 64 process subtypes available, where 0 is the default subtype for general use. The other subtypes are as follows:

  1 — 47 are reserved for definition by HP. Currently, 1 is a CMI process, 2 is a security monitor process, 30 is a device simulation process, and 31 is a spooler collector process. Also, for subtypes 1 to 15, PROCESS_LAUNCH_ rejects the create request with an invalid process subtype error unless the caller has a creator access ID of the super ID, or the program file is licensed, or the program file has the PROGID attribute set and an owner of the super ID.

  48 — 63 are for general use. Any user can create a named process with a device subtype in this range.

  For a list of all device types and subtypes, see Appendix A, Device Types and Subtypes.

- Reserved process names

  The operating system reserved process name space includes these names: $X*name*, $Y*name*, and $Z*name*, where *name* is from 1 through 4 alphanumeric characters. You should not use names of this form in any application. System-generated process names (from PROCESS_LAUNCH_, NEWPROCESS[NOWAIT], PROCESSNAME_CREATE_ , CREATEPROCESSNAME and CREATEREMOTENAME) are selected from this set of names. For more information about reserved process names, see Appendix B, Reserved Process Names.

- Creator access ID (CAID) and process access ID (PAID)

  The creator access ID of the new process is always the same as the process access ID of the creator process. The process access ID of the new process is the same as that of the creator process unless the program file has the PROGID attribute set; in that case the process access ID of the new process is the same as the user ID of the program file's owner and the new process is always local.

- I/O error to the home terminal

  An I/O error to the home terminal can occur if there are undefined externals in the program file and PROCESS_LAUNCH_ is unable to open or write to the home terminal to display the undefined externals messages. The *error-detail* parameter contains the file-system error number that resulted from the open or write that failed.

Nowait Considerations

- If you call this procedure in a nowait manner, the results are returned in the nowait PROCESS_LAUNCH_ or PROCESS_CREATE_ completion message (-102), not the output parameters of the procedure. The format of this completion message is described in the *Guardian Procedure Errors and Messages Manual*. If *error* is not 0, no completion message is sent to $RECEIVE. Errors can be reported either on return from the procedure, in which case *error* and *error-detail* might be meaningful, or through the completion message sent to $RECEIVE.

DEFINE Considerations

- DEFINEs are propagated to the new process from the process context of the caller, from a caller-supplied buffer containing DEFINEs collected by calls to DEFINESAVE, or from both of these. DEFINEs are propagated to the new process according to the DEFINE mode of the new process and the propagation option specified in Z^CREATE^OPTIONS. If both sets of DEFINEs are propagated and both sets contain a DEFINE with the same name, the DEFINE in the caller-supplied buffer is used. When a caller is creating its backup, the caller's DEFINEs are always propagated, regardless of the options chosen.

  The =_DEFAULTS DEFINE is always propagated, regardless of the options chosen. If the DEFINE buffer contains a =_DEFAULTS DEFINE, that one is

propagated; otherwise, the =_DEFAULTS DEFINE in the caller's context is propagated.

Buffer space for DEFINEs being propagated to a new process is limited to 2 MB whether the process is local or remote. However, the caller can propagate only as many DEFINEs as the child's PFS can accommodate in the buffer space for the DEFINEs themselves and in the operational buffer space needed to do the propagation. The maximum number of DEFINEs that can be propagated varies depending upon the size of the DEFINEs being passed.

- When a process is created, its DEFINE working set is initialized with the default attributes of CLASS MAP.
- The Z^PROGRAM^NAME, Z^LIBRARY^NAME, Z^SWAPFILE^NAME, or Z^EXTSWAPFILE^NAME fields can be DEFINE names; PROCESS_LAUNCH_ uses the disk volume or file given in the DEFINE. If Z^PROGRAM^NAME is a DEFINE name but no such DEFINE exists, an error is returned. If any of the other names is a DEFINE name but no such DEFINE exists, the procedure behaves as if no name were specified. This feature of accepting names of nonexistent DEFINEs as input gives the programmer a convenient mechanism that allows, but does not require, user specification of the location of the library file, the swap file, or the extended swap file.
- For each process, a count is kept of the changes to that process's DEFINEs. This count is always 0 for newly-created processes. The count is incremented each time the procedures DEFINEADD, DEFINEDELETE, DEFINESETMODE, and DEFINEDELETEALL are invoked and a consequent change to the process context occurs. In the case of DEFINEDELETE and DEFINEDELETEALL, the count is incremented by one even if more than one DEFINE is deleted. The count is also incremented if the DEFINE mode of the process is changed. If a call to CHECKDEFINE causes a DEFINE in the backup to be altered, deleted, or added, then the count for the backup process is incremented.

Batch Processing Considerations

Note. The job ancestor facility is intended for use by the NetBatch product. Other applications that use this facility might be incompatible with the NetBatch product.

- When the process being created is part of a batch job, PROCESS_LAUNCH_ sends a job process creation message to the job ancestor of the batch job. (See the discussion of "job ancestor" in the *Guardian Programmer's Guide*.) The message identifies the new process and contains the job ID as originally assigned by the job ancestor. This enables the job ancestor to keep track of all the processes belonging to a given job.

  For the format of the job process creation message, see the *Guardian Procedure Errors and Messages Manual*.
- PROCESS_LAUNCH_ can create a new process and establish that process as a member of the caller's batch job. In that case the caller's job ID is propagated to

the new process. If the caller is part of a batch job, to start a new process that is part of the caller's batch job, set Z^JOBID to -1.

- PROCESS_LAUNCH_ can create a new process separate from any batch job, even if the caller is a process that belongs to a batch job. In that case the job ID of the new process is 0. To start a new process that is not part of a batch job, specify 0 for Z^JOBID.
- PROCESS_LAUNCH_ can create a new batch job and establish the new process as a member of the newly created batch job. In that case, the caller becomes the job ancestor of the new job; the job ID supplied by the caller becomes the job ID of the new process. To start a new batch job, specify a nonzero value (other than -1) for Z^JOBID.

  A job ancestor must not have a process name that is greater than four characters (not counting the dollar sign). When the caller of PROCESS_LAUNCH_ is to become a job ancestor, it must conform to this requirement.
- When Z^JOBID is set to -1:
  - If the caller is not part of a batch job, neither is the newly created process; its job ID is 0.
  - If the caller is part of a batch job, the newly created process is part of the same job because its job ID is propagated to the new process.
- Once a process belongs to a batch job, it remains part of the job.

Safeguard Considerations

For information on processes protected by Safeguard, see the *Safeguard Reference Manual*.

OSS Considerations

- You cannot create an OSS process using the PROCESS_LAUNCH_ procedure. PROCESS_LAUNCH_ returns error 12 if you try. Use the PROCESS_SPAWN_ procedure or OSS functions to create an OSS process.
- You can call PROCESS_LAUNCH_ from an OSS process to create a Guardian process.
- Every Guardian process has these security-related attributes for accessing OSS objects. These attributes are passed, unchanged, from the caller to the new process, whether the caller is an OSS process or a Guardian process:
  - Real, effective, and saved user ID
  - Real, effective, and saved group ID
  - Group list
  - Login name

- Current working directory (cwd)
- Maximum file size
- Default OSS file security
- No other OSS process attribute is inherited by the new process.

- OSS file opens in the calling process are not propagated to the new process.

Related Programming Manuals

For programming information about the PROCESS_LAUNCH_ procedure, see the *Guardian Programmer's Guide*. For programming information on batch processing, see the appropriate NetBatch manual.

PROCESS_SETINFO_ Procedure

Summary

The PROCESS_SETINFO_ procedure alters a single nonstring attribute of a specified process and optionally returns the prior value of the attribute.

You can use the PROCESS_SETSTRINGINFO_ procedure to alter string-form process attributes such as home terminal.

Syntax for C Programmers

```
#include <cextdecs(PROCESS_SETINFO_)>

short PROCESS_SETINFO_ ( [ short *processhandle ]
                        ,[ short specifier ]
                        ,short set-attr-code
                        ,[ short *set-value ]
                        ,[ short set-value-len ]
                        ,[ short *old-value ]
                        ,[ short old-value-maxlen ]
                        ,[ short *old-value-len ] );
```

Syntax for TAL Programmers

```
error := PROCESS_SETINFO_ ( [ processhandle ]      ! i
                           ,[ specifier ]          ! i
                           ,set-attr-code          ! i
                           ,[ set-value ]          ! i
                           ,[ set-value-len ]      ! i
                           ,[ old-value ]          ! o
                           ,[ old-value-maxlen ]   ! i
                           ,[ old-value-len ] );   ! o
```

Parameters

error returned value

INT

returns a file-system error indicating the outcome of the operation.

processhandle input

INT .EXT:ref:10

is a process handle that specifies the process of interest. If this parameter is omitted or null, the caller is the process of interest. The null process handle is one which has -1 in each word (Refer to Guardian procedure call, PROCESSHANDLE_NULLIT_). However, PROCESS_SETINFO_ also treats a process handle with -1 in the first word as a null process handle.

specifier input

INT:value

indicates whether the operation should affect both members of a named process pair. Valid values are:

0 Act upon the specified process only.
1 Act upon both members of current instance of named process pair if *processhandle* specifies a member of a named process pair.

The default is 0.

Priority is the only attribute that can be altered for either a single member or both members of a named process pair. Changes to process file security and mom affect only a single process (*specifier* is treated as 0). Changes to item 49 (qualifier-info-available) always affect the named process pair as a whole (*specifier* is treated as 1).

set-attr-code input

INT:value

is the code specifying the process attribute to be altered. for more information about process attributes, see “Considerations” under this procedure and General Considerations on page 12-62.

set-value input

INT .EXT:ref:*

specifies the new value of the attribute to be altered.

set-value-len input

INT:value

specifies the length in words of *set-value*.

Both *set-value* and *set-value-len* must be present unless the attribute being altered is item 40 (mom’s-processhandle). In that case, both parameters can be omitted.

old-value output

INT .EXT:ref:*

if present and *old-value-maxlen* is not 0, returns the prior value of the attribute being altered. When this parameter is present, *old-value-maxlen* and *old-value-len* must be present; otherwise, all three parameters must be omitted.

If priority is being altered for both members of a process pair, the old priority of the primary process is returned.

old-value-maxlen input

INT:value

specifies the length in words of the variable *old-value*. This parameter must be present when *old-value* is present; otherwise, it must be omitted.

old-value-len output

INT .EXT:ref:1

is the actual length in words of *old-value*. This parameter must be present when *old-value* is present; otherwise, it must be omitted.

set-attr-code

set-attr-code and its associated value can be one of these attributes:

Attribute Code		TAL Value Representation
40	mom's process handle	INT (10 words)
* 41	process file security	INT
42	priority	INT
+* 45	logged-on state	INT
* 47	primary	INT
* 49	qualifier information available	INT
+* 50	Safeguard-authenticated logon	INT
+* 69	stop on logoff	INT
+* 70	propagate logon	INT
+* 71	propagate stop-on-logoff	INT
75	nice() caller's value	INT(32)
*104	maximum size of the main stack	INT(32)

Attributes marked with an asterisk (*) can be altered only when the caller is the target process. Attributes marked with a plus sign (+) can be altered only when the caller is a privileged process.

- 40: mom's process handle

  sets the process handle of the mom of the specified process. The process handle to be used must only be the process handle of the calling process. If the calling process attempts to specify a process handle other than itself as the mom process handle, PROCESS_SETINFO_ ignores that parameter. See Considerations for more information.

- 41: process file security

  sets the current default process file security setting. The security bits are as follows:

```
<0:3>     0
<4:6>     ID code allowed for read
<7:9>     ID code allowed for write
<10:12>   ID code allowed for execute
<13:15>   ID code allowed for purge
```

ID code can be one of these:

0 Any user (local)
1 Member of owner's group (local)
2 Owner (local)
4 Any user (local or remote)
5 Member of owner's community (local or remote)
6 Owner (local or remote)
7 Super ID only (local)

- 42: current priority

  sets execution priority to be assigned to the new process. Execution priority is a value in the range of 1 to 199, where 199 is the highest possible priority.

- 45: logged-on state

  sets information about the logged-on state of the process. The fields are:

  ```
  <0:14>        Reserved (specify 0)

    <15>    0   Process is not logged on.
            1   Process is logged on.
  ```

- 47: primary

  specify 1 if the process is the current primary of a named process pair, 0 otherwise.

- 49: qualifier information available

  specify 1 if the process is prepared to respond to the subordinate name inquiry system message (-107). This message is received when another process calls the FILENAME_FINDNEXT_ procedure to obtain qualifier names of the process. If the process does not respond to the message, other processes calling the FILENAME_FINDNEXT_ procedure on the system might be blocked.

  Specify 0 to disable the receipt of the subordinate name inquiry system message (-107).

  For the format of the subordinate name inquiry message, see the *Guardian Procedure Errors and Messages Manual*.

- 50: Safeguard-authenticated logon

  specify 1 if a Safeguard-authenticated logon has taken place (that is, if the process was started after successfully logging on a through terminal owned by Safeguard), 0 otherwise. This information can be set for the caller only.

- 69: stop on logoff

specify 1 if the process is to be stopped when it requests to be placed in the logged-off state, 0 otherwise.

- 70: propagate logon

  specify 1 if the process's local descendants are to be created with the inherited-logon flag set, 0 otherwise.

- 71: propagate stop-on-logoff

  specify 1 if the process's local descendants are to be created with the stop-on-logoff flag set, 0 otherwise.

- 75: nice() caller's value

  Because the nice() value is INT(32), *set-value-len* must be two words.

- 104: maximum size of the main stack

  sets the maximum main stack size in bytes. You cannot specify a value that is less than the current main stack size or greater than the system will allow (32 megabytes (MB)). To obtain the current main stack size, call the PROCESS_GETINFOLIST_ procedure with the current main stack size attribute (103).

Considerations

- The caller of PROCESS_SETINFO_

  When PROCESS_SETINFO_ is called on a Guardian process, the caller must be the super ID, the group manager of the process access ID, or a process with the same process access ID as the process or process pair whose attribute is being changed. For information about the process access ID, see the description under "Considerations" for the PROCESS_GETINFO_ procedure and the *Guardian User's Guide*.

  When PROCESS_SETINFO_ is called on an OSS process, the security rules that apply are the same as those that apply when calling the OSS `kill()` function. See the `kill(2)` function reference pages either online or in the *Open System Services System Calls Reference Manual* for details.

  The caller must be local to the same system as the specified process. A process is considered to be local to the system on which its creator is local. A process is considered to be remote, even if it is running on the local system, if its creator is remote. (In the same manner, a process running on the local system whose creator is also running on the local system might still be considered remote because its creator's creator is remote.)

  A remote process running on the local system can become a local process by successfully logging on to the local system using a call to the USER_AUTHENTICATE_ procedure (or VERIFYUSER). After a process logs on to the local system, any processes that it creates are considered local.

- Attributes that can be set or cleared by privileged callers only

  Several attributes can be set or cleared (set to 0) by privileged callers only, as follows:

item 45 (logged-on state)	must be priv to set, nonpriv can clear
item 50 (Safeguard-authenticated logon)	must be priv to set or clear
item 70 (propagate logon)	must be priv to set, nonpriv can clear
item 75 (nice() caller's value)	must be priv to set or clear

- Mom's process handle

  Specifying item 40 (mom's process handle) is analogous to calling STEPMOM on a process. The caller becomes the new mom of the specified process, and will receive the process deletion system message when the process terminates. The former mom will not receive a process deletion system message. *specifier* is ignored, as this operation applies only to a single target process. *set-value* is ignored, because the caller must be doing the adoption on its own behalf; no third-party adoptions are permitted.

  A process should not alter the mom of a member of a named process pair by calling either PROCESS_SETINFO_ or STEPMOM. This causes errors and interferes with operation, as correct operation depends upon each member of a named process pair being the other member's mom.

  The creator of a named process should not adopt its named child process, even if the child is a single named process rather than a named process pair. Doing so establishes the creator as its mom as well as its ancestor. When the process terminates, the creator will receive two system messages—one for the disappearance of the named process as an entity, and one for the disappearance of the adopted process.

  If PROCESS_SETINFO_ is used to set the mom of an OSS process, the new mom receives the Guardian process deletion message when the OSS process terminates. The received message contains an indication that the terminated process was an OSS process and also contains the OSS process ID; otherwise, the message is the same as one received for a terminating Guardian process. For more information on the Guardian parent of an OSS process, see Keeping Track of OSS Child Processes on page 12-183.

  If the OSS process successfully executes a function from the `exec` or `tdm_exec` set of functions, a Guardian process deletion message is sent to the mom. Although the process is still alive in the OSS environment (the OSS process ID still exists), the process handle no longer exists, so the process has terminated in the Guardian environment.

  The OSS parent process (which is not necessarily the same process as the mom process) also receives OSS process termination status if the OSS process ID no longer exists. The order of delivery of the OSS process termination status and the Guardian process deletion message is not guaranteed.

See the *Guardian Procedure Errors and Messages Manual* for the format of the Guardian process deletion message. See the `wait(2)` function reference pages either online or in the or the *Open System Services System Calls Reference Manual* for details on the OSS process termination status.

- Priority

  A process has two priority values: the initial priority and the current priority. Specifying item 42 (priority) causes the initial priority to be changed to the specified new value. The current priority is updated to the initial priority when the process waits for an external event to occur.

  Although PROCESS_SETINFO_ supersedes PRIORITY, it does not return the initial priority value. Initial priority can be obtained by calling PROCESS_GETINFOLIST_.

- Primary

  If a switch or a backup takeover occurs (causing the backup process to become the new primary) through use of the checkpoint procedures, it is not necessary to use PROCESS_SETINFO_ to set the primary attribute of the new primary. The checkpoint procedures automatically identify the new primary process to the operating system.

Safeguard Considerations

For information on processes protected by Safeguard, see the *Safeguard Reference Manual.*

OSS Considerations

If PROCESS_SETINFO_ is used to change the priority of an OSS process, the same security rules apply as for the OSS `kill()` function. See the `kill(2)` function reference pages either online or in the *Open System Services System Calls Reference Manual* for details.

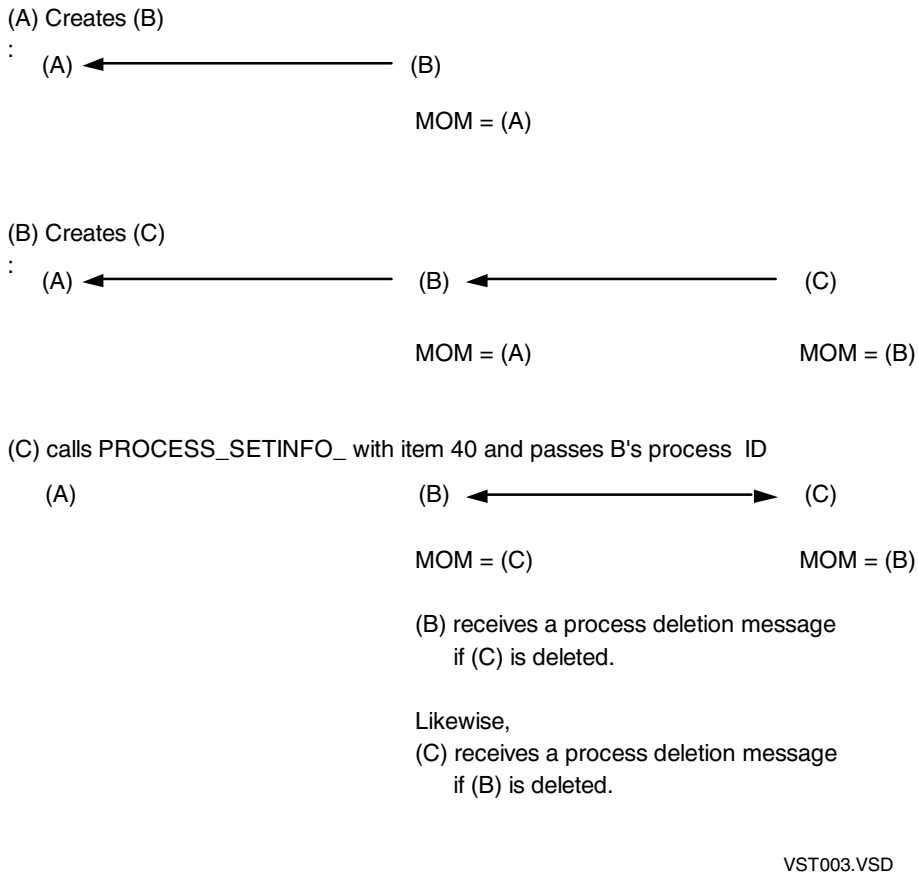
Example

```
INT set^attribute^code := 42;   ! set execution priority
      .
      .
err := PROCESS_SETINFO_ ( proc^handle , ,
                          set^attribute^code , new^priority ,
                          set^value^length );
```

Related Programming Manual

For programming information about the PROCESS_SETINFO_ procedure, see the *Guardian Programmer's Guide*.

Figure 12-1. Effect of Adopting a Process



PROCESS_SETSTRINGINFO_ Procedure

Summary

The PROCESS_SETSTRINGINFO_ procedure alters a single string-form attribute of a specified process, and optionally returns the prior value of the attribute.

You can use the PROCESS_SETINFO_ procedure to alter nonstring process attributes.

Syntax for C Programmers

```
#include <cextdecs(PROCESS_SETSTRINGINFO_)>

short PROCESS_SETSTRINGINFO_ ( [ short *processhandle ]
                              ,[ short specifier ]
                              ,short set-attr-code
                              ,const char *set-value
                              ,short length
                              ,[ char *old-value ]
                              ,[ short maxlen ]
                              ,[ short *old-value-len ] );
```

- The parameter *length* specifies the length in bytes of the character string pointed to by *set-value*. The parameters *set-value* and *length* must either both be supplied or both be absent.
- The parameter *maxlen* specifies the maximum length in bytes of the character string pointed to by *old-value*, the actual length of which is returned by *old-value-len*. All three of these parameters must either be supplied or be absent.

Syntax for TAL Programmers

```
error := PROCESS_SETSTRINGINFO_ ( [ processhandle ]        ! i
                                 ,[ specifier ]            ! i
                                 ,set-attr-code            ! i
                                 ,set-value:length         !
i:i
                                 ,[ old-value:maxlen ]     !
o:i
                                 ,[ old-value-len ] );     ! o
```

Parameters

error returned value

INT

returns a file-system error number indicating the outcome of the operation.

processhandle input

INT .EXT:ref:10

is a process handle that specifies the process of interest. If this parameter is omitted or null, the caller is the process of interest. The null process handle is one which has -1 in each word (Refer to Guardian procedure call, PROCESSHANDLE_NULLIT_). However, PROCESS_SETSTRINGINFO_ also treats a process handle with -1 in the first word as a null process handle.

specifier input

INT:value

indicates whether the operation should affect both members of a named process pair. Valid values are:

0 Act upon the specified process only.
1 Act upon both members of current instance of named process pair if *processhandle* specifies a member of a named process pair.

The default is 0.

A change to the home terminal affects only a single process (*specifier* is treated as 0).

set-attr-code input

INT:value

is the code specifying the process attribute to be altered. For more information about process attributes, see “Considerations” under this procedure and under PROCESS_GETINFO_ Procedure .

set-value:*length* input:input

STRING .EXT:ref:*, INT:value

is the new value for the attribute to be altered. The value of *set-value* must be exactly *length* bytes long.

old-value:*maxlen* output:input

STRING .EXT:ref:*, INT:value

if present and *maxlen* is not 0, returns the prior value for the attribute being altered.

old-value-len output

INT .EXT:ref:1

is the actual length in bytes of *old-value*. This parameter must be present when *old-value* is present. Otherwise, both must be omitted.

Considerations

- The caller of PROCESS_SETSTRINGINFO_ must be the super ID, the group manager of the process access ID, or a process with the same process access ID as the process or process pair whose attribute is being changed. For information about the process access ID, see General Considerations on page 12-62 and the *Guardian User’s Guide*.

The caller must be local to the same system as the specified process. A process is considered to be local to the system on which its creator is local. A process is considered to be remote, even if it is running on the local system, if its creator is remote. (In the same manner, a process running on the local system whose creator is also running on the local system might still be considered remote because it's creator's creator is remote.)

A remote process running on the local system can become a local process by successfully logging on to the local system by a call to USER_AUTHENTICATE_ (or VERIFYUSER). After a process logs on to the local system, processes that it creates are considered local.

- *set-attr-code*

  set-attr-code can be one of these attributes:

  ```
  *  5 = home terminal
  ```

  An asterisk (*) indicates that the attribute can be altered only when the caller is the target process.

- home terminal

  This is the only attribute that currently can be altered by PROCESS_SETSTRINGINFO_. If a process alters this attribute, the new home terminal becomes the home terminal for any process that it subsequently creates.

Example

```
attr^code := 5;     ! alter home terminal name
error := PROCESS_SETSTRINGINFO_ ( , , attr^code,
                                  new^termname:namelen );
```

Related Programming Manual

For programming information about the PROCESS_SETSTRINGINFO_ procedure, see the *Guardian Programmer's Guide*.

PROCESS_SPAWN_ Procedure

Summary

Note. The TAL or pTAL syntax for this procedure is declared only in the EXTDECS0 file.

The PROCESS_SPAWN_ procedure creates a new Open System Services (OSS) process and, optionally, assigns a number of process attributes. You can use this procedure to create only OSS processes, although you can call it from either a Guardian process or an OSS process. To create a Guardian process, call the PROCESS_LAUNCH_ procedure.

DEFINEs can be propagated to the new process. The DEFINEs can come from the caller's context or from a buffer of DEFINEs saved by the DEFINESAVE procedure.

PROCESS_SPAWN_ differs from the OSS functions that create OSS processes in these ways:

- You can specify that the new process be created in either a waited or nowait manner. When it is created in a waited manner, identification for the new process is returned directly to the caller. When it is created in a nowait manner, its identification is returned in a system message sent to the caller's $RECEIVE file.
- You can obtain a level of fault tolerance in OSS processes by calling PROCESS_SPAWN_ to create OSS processes from a monitor implemented as a Guardian process pair. The monitor checks that the created OSS process continues to run and restarts it if there is a failure. For more information on writing fault-tolerant programs, see the *Guardian Programmer's Guide*.
- You can call PROCESS_SPAWN_ from either a Guardian process or an OSS process.
- The caller of PROCESS_SPAWN_ becomes the Guardian parent of the new OSS process.
- Because the caller of PROCESS_SPAWN_ is the Guardian parent of the new OSS process, when the new process is terminated, it receives a process deletion system message (-101) through its $RECEIVE file rather than an OSS `SIGCHLD` signal. The caller also receives this message (with a different completion code) when the child process calls one of the OSS `exec` set of functions and migrates to

a new process handle. For more information on the process deletion message and its completion codes, see the *Guardian Procedure Errors and Messages Manual*.

- The new process does not have an OSS caller; instead it is considered to be an OSS orphan process with a caller process ID of 1.
- OSS file opens in the calling process are not propagated to the new process. The file opens must be specified explicitly in the *fdinfo* parameter.
- The created OSS process is always the leader of its own session.
- The calling process is not required to be compliant with the Common Run-Time Environment.
- PROCESS_SPAWN_ can create a process on local system or on a remote system.
- These following OSS attributes are passed, unchanged, from the caller to the new OSS process, whether the caller is an OSS process or a Guardian process:
  - Real, effective, and saved OSS user ID
  - Real, effective, and saved group ID
  - Group list
  - Login name
  - Current working directory
  - Maximum file size
  - Default OSS file security

  No other OSS process attribute is inherited by the new process.

For more information on creating an OSS process and for details on the parameters to this procedure, see the `tdm_spawn()` function reference page either online or in the *Open System Services System Calls Reference Manual*.

Syntax for C Programmers

```
#include <cextdecs(PROCESS_SPAWN_)>

__int32_t PROCESS_SPAWN_ ( [ char *oss-program-file ]
                          ,[ void *fdinfo ]
                          ,[ char *argv ]
                          ,[ char *envp ]
                          ,[ void *inheritance ]
                          ,[ __int32_t inheritance-length ]
                          ,[ void *process-extension ]
                          ,[ void *process-results ]
                          ,[ __int32_t nowait-tag ]
                           ,[ char *path ] );
```

- CEXTDECS (through the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
oss-pid:= PROCESS_SPAWN_ ( oss-program-file          ! i
                          ,[ fdinfo ]                ! i
                          ,[ argv ]                  ! i
                          ,[ envp ]                  ! i
                          ,[ inheritance ]           ! i
                          ,[ inheritance-length ]    ! i
                          ,[ process-extension ]     ! i
                          ,[ process-results ]       ! i:o
                          ,[ nowait-tag ]            ! i
                          ,[ path ] );               ! i
```

Parameters

oss-pid returned value

INT(32) .EXT:ref:1

returns the OSS process ID of the new process. If you created the process in a nowait manner, then the returned value is 0D and the OSS process ID is returned in the completion message sent to $RECEIVE. If an error occurs, the returned value is -1D.

oss-program-file input

STRING .EXT:ref:*

specifies the null-terminated OSS pathname of the OSS program file to be run. If the pathname is an absolute pathname, it is resolved relative to the root of the caller. If the pathname is a relative pathname, it is resolved with respect to the

caller's current working directory. If the pathname is the program name, the path provided in the *path* parameter is searched for the program file.

Shell scripts that exist on nodes other than the caller's node (remote shell scripts) cannot be spawned (for more information, see Considerations for Resolving File Names on page 12-182). Shell scripts that exist on the caller's node (local shell scripts) are supported, but security is ignored if an interpreter that exists on another node is used. A shell script must contain this string syntax in the first line of the file when the *path* parameter is not specified:

```
#! interpreter-name optional-arguments
```

If the Guardian caller does not already have a current working directory, PROCESS_SPAWN_ attempts to establish the caller's default subvolume as the current working directory.

For a description of OSS pathname syntax, see Appendix D, File Names and Process Identifiers.

fdinfo input

STRING .EXT:ref:(ZSYS^DDL^FDINFO)

specifies the file creation mask, current working directory, and file descriptors to be opened or duplicated by the new process. This parameter also allows the caller to limit the time allowed for the child process to open all of its files. If the pathnames are absolute pathnames, they are resolved relative to the child's root. If the pathnames are relative pathnames, they are resolved relative to the child's current working directory. For information on how to assign field values to the structure, see "Structure Definition for *fdinfo*."

argv input

EXTADDR .EXT:ref:1

if present and not equal to 0D, specifies the address of an array of addresses that point to null-terminated strings containing arguments to be passed to the main function of the new process. The last member of this array must be a null pointer (0D). Most programs expect *argv*[0] to point to a null-terminated string containing the pathname of the OSS program file (use the address of the *oss-program-file* parameter). The argument (*argv*) string is passed to the child unmodified by PROCESS_SPAWN_.

The number of bytes available for the new process's combined argument (*argv*) and environment (*envp*) lists has a system-imposed limit. This limit, which includes the pointers and the null terminators on the strings, is available by calling the OSS sysconf(_SC_ARG_MAX) function.

envp input

EXTADDR .EXT:ref:1

if present and not equal to 0D, specifies the address of an array of addresses that point to null-terminated strings that describe the environment of the new process. The last member of this array must be a null pointer (0D). The environment (*envp*) string is passed to the child unmodified by the PROCESS_SPAWN_ procedure. Most programs expect these strings to have this syntax:

```
name = value
```

The number of bytes available for the new process's combined argument (*argv*) and environment (*envp*) lists has a system-imposed limit. This limit, which includes the pointers and the null terminators on the strings, is available by calling the OSS sysconf(_SC_ARG_MAX) function.

inheritance input

STRING .EXT:ref:(ZSYS^DDL^INHERITANCE)

if *inheritance-length* is not zero, specifies which signals are either blocked or use default action for the new process. For information on how to assign field values to the structure, see "Structure Definition for *inheritance*."

inheritance-length input

INT(32):value

specifies the length in bytes of *inheritance*. This parameter is required if *inheritance* is specified.

process-extension input

STRING .EXT:ref:(ZSYS^DDL^PROCESSEXTENSION)

specifies the Guardian attributes of the new process. For information on how to set the field values of the structure, see "Structure Definition for *process-extension*."

process-results input:output

STRING .EXT:ref:(ZSYS^DDL^PROCESSRESULTS)

provides Guardian information on the outcome of the procedure call. For information on the field values of the structure, see "Structure Definition for *process-results*."

The tdmext.h header file is not kept current when new error codes are defined for process creation

functions. The list of _TPC_ macros described in this reference page is not complete; for a current description of error macros and error codes, see the Guardian header file $SYSTEM.ZSPIDEF.ZGRDC or Table 12-3, Summary of Process Creation Errors .

nowait-tag input

INT(32):value

if present and not equal to -1D, indicates that the process is to be created in a nowait manner; the PROCESS_SPAWN_ procedure returns as soon as process creation is initiated. For details ,see Nowait Considerations on page 12-142. The Z^TIMEOUT field of the *fdinfo* structure also contains nowait considerations.

If *nowait-tag* is equal to -1D, the process is created in a waited manner.

path input

STRING .EXT:ref:*

if present and not null (0D), and if the *oss-program-file* parameter does not contain a slash character, specifies a null-terminated string of path prefixes separated by colons to further identify the *oss-program-file* parameter. If the resolved name is not the name of a program file, then the *oss-program-file* parameter is treated as a shell script for the command interpreter. The path parameter is equivalent to the OSS path environment variable.

If the path parameter is not specified, then a shell script must contain this string syntax in the first line of the file:

```
#! interpreter-name optional-arguments
```

Specifically, if the *path* parameter is not specified or is null (0D), and a shell script does not have the syntax shown above, PROCESS_SPAWN_ returns OSS errno value 4008 (ENOEXEC) in the ZSYS-DDL-PROCESSRESULTS.Z-ERRNO field of the *process-results* parameter.

Note. Use only the files specified in these pages to obtain the definitions for the structures and literals. The definitions in other files may produce undesired results.

Structure Definition for *fdinfo*

The *fdinfo* parameter specifies the file descriptors to be opened or duplicated by the new process.

The structure for the *fdinfo* parameter can contain multiple occurrences of the Z^FDENTRY substructure (fdentry for C programs).

In the TAL ZSYSTAL file, the structure for the *fdinfo* parameter is defined as:

```
STRUCT ZSYS^DDL^FDINFO^DEF (*);
   BEGIN
   INT(32) Z^LEN;
   INT(32) Z^TIMEOUT;
   INT(32) Z^UMASK;
   INT(32) Z^CWD;
   INT(32) Z^FDCOUNT;
   STRUCT Z^FDENTRY;
      BEGIN
      INT(32) Z^FD;
      INT(32) Z^DUPFD;
      INT(32) Z^NAME;
      INT(32) Z^OFLAG;
      INT(32) Z^MODE;
      END;
   END;
```

For TAL programs, these default values must be specified when an option is not wanted:

Field Name	Default Value
Z^LEN	$OFFSET(ZSYS^DDL^FDINFO.FDENTRY.Z^MODE) + $LEN(ZSYS^DDL^FDINFO.FDENTRY.Z^MODE)
Z^TIMEOUT	-1D
Z^UMASK	-1D
Z^CWD	0D
Z^FDCOUNT	0D
Z^FDENTRY.Z^FD	0D
Z^FDENTRY.Z^DUPFD	0D
Z^FDENTRY.Z^NAME	0D
Z^FDENTRY.Z^OFLAG	0D
Z^FDENTRY.Z^MODE	0D

In the C tdmext.h header file, the structures for the *fdinfo* parameter are defined as:

```
typedef struct fdinfo {
   long  z_len;
   long  z_timeout;
   long  z_umask;
   char *z_cwd;
   long  z_fdcount;
   fdentry_def z_fdentry;
} fdinfo_def;

typedef struct fdentry {
   long  z_fd;
   long  z_dupfd;
   char *z_name;
   long  z_oflag;
   long  z_mode;
} fdentry_def;
```

C programs should initialize the fdinfo structure by using the #define DEFAULT_FDINFO in the tdmext.h header file.

`Z^Len`

is the length of the ZSYS^DDL^FDINFO structure including one occurrence of the Z^FDENTRY substructure. Because the structure is subject to change, Z^LEN is used by PROCESS_SPAWN_ to identify the version of the structure.

`Z^Timeout`

indicates how long the new process waits for the OSS `open()` and `dup()` functions to complete opening and duplicating all files specified in Z^FDENTRY. Z^TIMEOUT can have these values:

- > 0D specifies a wait-for-completion. The value specifies the maximum time (in 1-second units) that the new process can wait for completion of the OSS `open()` and `dup()` functions. If PROCESS_SPAWN_ is called in a nowait manner, the completion message is sent when the Z^TIMEOUT value is reached or when all files are opened or duplicated, whichever comes first.
- = -1D specifies an indefinite wait. If PROCESS_SPAWN_ is called in a nowait manner, the completion message is sent when all files are opened or duplicated.
- = 0D specifies a return after the new process is created. After the new process is created, PROCESS_SPAWN_ returns immediately to the caller, regardless of whether open or duplication completions occur. If PROCESS_SPAWN_ is called in a nowait manner, the completion message is sent when the new process is created.

Z^Umask

is the OSS file mode creation mask of the new process. A value of -1 indicates that the OSS file mode creation mask of the calling process should be used. For more information see the umask() function reference page either online or in the *Open System Services System Calls Reference Manual*.

Z^Cwd

is the address of a string containing the null-terminated OSS pathname of the OSS current working directory of the new process. A value of 0D indicates that the OSS current working directory of the calling process should be used. If the caller does not have a current working directory, then the caller's default volume is used. An absolute pathname should be specified, because relative file names are resolved using the undefined environment of the new process.

Z^FdCount

is the number of Z^FDENTRY substructures occurring in the structure. Each substructure specifies a file descriptor to be opened or duplicated by the new process.

Z^FdEntry

describes a file descriptor in these fields.

Z^Fd

is the file descriptor to be opened. Z^FD can have these values:

0D	standard input
1D	standard output
2D	standard error
other	user-defined

Z^DUPFd

indicates whether the file descriptor specified in Z^FD is to be opened as a duplicate with the OSS dup() function. Z^DUPFD can have these values:

> 0D This file descriptor is a duplicate of a file descriptor previously specified in Z^FD.

= -1D This file descriptor is not a duplicate. Open the file descriptor with the OSS open() function according to the values in these fields.

Z^Name

is the address of a string containing the null-terminated OSS pathname of the file to be opened by the new process. It must be possible to open this file with the OSS open() function. A relative pathname is resolved with the value for the OSS current working directory in Z^CWD.

To have the new process open a pipe, specify a named pipe (also known as a FIFO) in the Z^NAME field. To create a FIFO, use the OSS `mkfifo()` function.

`Z^Oflag`

is the file access flag and file status flag to be used by the OSS `open()` function called by the new process. This field is ignored when the file is opened as a duplicate. For more information on these flags, see the `open()` reference page either online or in the *Open System Services System Calls Reference Manual*.

These tables summarize the values for Z^OFLAG. TAL literals are defined in the ZSYSTAL file. Literals in the zsysc file, for C programs, are the same as those for TAL except that they contain the underscore (_) character instead of the circumflex (^) character.

One of these file access flags must be supplied:

Name (ZSYS^VAL^)	Value	Description	Corresponding `open()` Flag
OSSOPEN^RDONLY	0	Open only for reading	O_RDONLY
OSSOPEN^RDWR	2	Open for reading and writing	O_RDWR
OSSOPEN^WRONLY	1	Open only for writing	O_WRONLY

One of these file status flags can be supplied:

Name (ZSYS^VAL^)	Value	Description	Corresponding `open()` Flag
OSSOPEN^APPEND	4	Open only for append access	O_APPEND
OSSOPEN^CREAT	8	Create and open the file	O_CREAT
OSSOPEN^EXCL	32	Open in exclusive access mode	O_EXCL
OSSOPEN^NOCTTY	32768	Do not open as controlling terminal	O_NOCTTY
OSSOPEN^NONBLOCK	16384	Open for nonblocked access	O_NONBLOCK
OSSOPEN^SYNC	65536	Open for synchronized update	O_SYNC
OSSOPEN^TRUNC	16	Open and empty the file	O_TRUNC

Z^MODE

is the read, write, and execute permissions of the file to be created when Z^OFLAG is set to ZSYS^VAL^OSSOPEN^CREAT. Otherwise, Z^MODE is ignored. For more information on file permissions, see the OSS `open()` reference page either online or in the *Open System Services System Calls Reference Manual*.

Structure Definition for *inheritance*

The *inheritance* parameter specifies which signals are blocked or use default action for the new process. For more information on OSS signals, see the signal(4) reference page either online or in the *Open System Services System Calls Reference Manual*.

In the TAL ZSYSTAL file, the structure for the *inheritance* parameter is defined as:

```
STRUCT ZSYS^DDL^INHERITANCE^DEF (*);
   BEGIN
   INT     Z^FLAGS;
   INT     Z^FILLER;
   INT(32) Z^PGROUP;
   INT(32) Z^SIGMASK;
   INT(32) Z^SIGDEFAULT;
   END;
```

For TAL programs, this default value must be specified when this parameter is not wanted:

Field Name	Default Value
Z^FLAGS	0

In the C spawnh file, the structure for the *inheritance* parameter is defined as:

```
typedef struct inheritance {
   short     flags;
#define SPAWN_SETGROUP 0x01  /* not used by PROCESS_SPAWN_ */
#define SPAWN_SETSIGMASK 0x02/* controls child sigmask*/
#define SPAWN_SETSIGDEF 0x04 /* controls child sigmask*/
#define SPAWN_NOTDEFD 0xFFF8 /* undefined bit fields  */
   char      filler_1[2];
   pid_t     pgroup;
   sigset_t  sigmask;
   sigset_t  sigdefault;
} inheritance;
```

C programs should initialize the inheritance structure by setting the flags field to 0.

Z^Flags

specifies whether the Z^SIGMASK field or the Z^SIGDEFAULT field or both fields are specified.

These tables summarize the settings for Z^FLAGS. TAL literals are defined in the ZSYSTAL file. Literals in the zsysc file, for C programs, are the same as those for TAL except that they contain the underscore (_) character instead of the circumflex (^) character.

Either one or both of these values can be supplied:

`Z^Pgroup`

is not used by PROCESS_SPAWN_. Specify 0D.

`Z^SigMask`

is a mask indicating which signals are to be blocked by the new process when Z^FLAGS contains ZSYS^VAL^SPAWN^SETSIGMASK. When Z^SIGMASK.<*n*> is set to 1, the signal represented by bit <*n*> is blocked.

Name (ZSYS^VAL^)	Value	Description
SPAWN^SETSIGDEF	4	indicates that Z^SIGDEFAULT is specified
SPAWN^SETSIGMASK	2	indicates that Z^SIGMASK is specified

`Z^SigDefault`

is a mask indicating which signals are to use default action for the new process when Z^FLAGS contains ZSYS^VAL^SPAWN^SETSIGDEF. When Z^SIGDEFAULT.<*n*> is set to 1, the signal represented by bit <*n*> is used.

Structure Definition for *process-extension*

The *process-extension* parameter specifies the Guardian attributes of the new process.

In the TAL ZSYSTAL file, the structure for the *process-extension* parameter is defined as:

```
STRUCT ZSYS^DDL^PROCESSEXTENSION^DEF (*);
   BEGIN
   INT(32) Z^LEN;
   INT(32) Z^LIBRARYNAME;
   INT(32) Z^SWAPFILENAME;
   INT(32) Z^EXTSWAPFILENAME;
   INT     Z^PRIORITY;
   INT     Z^CPU;
   INT     Z^NAMEOPTIONS;
   INT     Z^FILLER;
   INT(32) Z^PROCESSNAME;
   INT(32) Z^HOMETERM;
   INT     Z^MEMORYPAGES;
   INT     Z^JOBID;
   INT     Z^CREATEOPTIONS;
   INT     Z^FILLER1;
   INT(32) Z^DEFINES;
   INT     Z^DEFINESLEN;
   INT     Z^DEBUGOPTIONS;
   INT(32) Z^PFSSIZE;
   INT     Z^OSSOPTIONS;
   INT     Z^FILLER2;
   INT(32) Z^MAINSTACKMAX;
   INT(32) Z^HEAPMAX;
   INT(32) Z^SPACEGUARANTEE;
   END;
```

For TAL programs, these default values must be specified when an option is not wanted:

Field Name	Default Value
Z^LEN	$OFFSET (ZSYS^DDL^PROCESSEXTENSION.Z^OSSOPTIONS) + $LEN (ZSYS^DDL^PROCESSEXTENSION.Z^OSSOPTIONS) + 2
Z^LIBRARYNAME	0D
Z^SWAPFILENAME	0D
Z^EXTSWAPFILENAME	0D
Z^PRIORITY	-1
Z^CPU	-1
Z^NAMEOPTIONS	ZSYS^VAL^PCREATOPT^NONAME
Z^PROCESSNAME	0D
Z^HOMETERM	0D
Z^MEMORYPAGES	-1
Z^JOBID	-1
Z^CREATEOPTIONS	ZSYS^VAL^PCREATOPT^DEFAULT
Z^DEFINES	0D
Z^DEFINESLEN	0
Z^DEBUGOPTIONS	ZSYS^VAL^PCREATOPT^DEFAULT
Z^PFSSIZE	0
Z^OSSOPTIONS	ZSYS^VAL^PSPAWNOPT^DEFAULT
Z^MAINSTACKMAX	0D
Z^HEAPMAX	0D
Z^SPACEGUARANTEE	0D

In the C tdmext.h header file, the structure for the *process-extension* parameter is defined as:

```
typedef struct process_extension {
   long    pe_len;
   char   *pe_library_name;
   char   *pe_swap_file_name;
   char   *pe_extswap_file_name;
   short   pe_priority;
   short   pe_cpu;
   short   pe_name_options;
   char    filler_1[2];
   char   *pe_process_name;
   char   *pe_hometerm;
   short   pe_memory_pages;
   short   pe_jobid;
   short   pe_create_options;
   char    filler_2[2];
   char   *pe_defines;
   short   pe_defines_len;
   short   pe_debug_options;
   long    pe_pfs_size;
   short   pe_OSS_options;
   char    filler_3[2];
   long    z_mainstackmax;
   long    z_heapmax;
   long    z_spaceguarantee;
} process_extension_def;
```

C programs should initialize the process_extension structure by using the #define DEFAULT_PROCESS_EXTENSION in the tdmext.h header file.

`Z^Len`

is the length of the ZSYS^DDL^PROCESSEXTENSION structure. Because the structure is subject to change, Z^LEN is used by PROCESS_SPAWN_ to identify the version of the structure.

`Z^LibraryName`

is the address of the null-terminated OSS pathname of the Guardian user library file to be used by the new process. For the program to create a linkage to the library file, the caller must have write permission to the program file. If the pathname is relative, it is resolved using the OSS current working directory.

If the Z^LIBRARYNAME field of the *process-extension* parameter is specified, external references are resolved first from the specified Z^LIBRARYNAME, then from the system library.

If you specify the value -1D, the new process is to run with no user library file. (The references that were previously resolved from the user library are resolved from the system library.) For the program to remove a linkage to a library file, the caller must have write permission to the program file.

If you specify the default value 0D, the process uses the same user library file (if any) as it did the last time it was run (or with no file if that was how it was run) or with the library file currently executing. Write permission to the program file is not required. For more information about TNS user libraries, see the *Binder Manual*. For more information about TNS/R native user libraries and shared run-time libraries, see the *nld and noft Manual*. For more information about dynamic-link libraries (including native user libraries used with PIC programs), see the *ld and rld Reference Manual*.

If an external reference cannot be resolved, it is modified to invoke the debugger when referenced. PROCESS_SPAWN_ then returns a warning 14 in the Z^TPCERROR field of the *process-results* parameter and issues a warning message to the home terminal the first time the program is run. (The warning 14 and the terminal message are issued again the first time the program is run following a system load).

`Z^SwapFileName`

is not used, but you can provide it for informational purposes. If supplied, the swap file must be on the same system as the process being created. If the supplied name is in local form, the system where the process is created is assumed. Processes swap to a file that is managed by the Kernel-Managed Swap Facility. For more information on this facility, see the *Kernel-Managed Swap Facility (KMSF) Manual*. To reserve swap space for the process, specify the Z^SPACEGUARANTEE field. Alternatively, use the `nld` utility to set TNS/R native process attributes or the `eld` utility to set TNS/E native process attributes.

For TNS processes on RVUs preceding the D42 RVU, this parameter is the address of the null-terminated OSS pathname of the Guardian swap file to be used for the user data stack segment of the process. The swap file must be an unstructured file. If the pathname is relative, it is resolved using the OSS current working directory.

The default value is 0D.

For more information about the swap files, see Considerations for Resolving File Names on page 12-182 and DEFINE Considerations on page 12-142.

`Z^ExtSwapFileName`

for TNS processes, if set to the default value, 0D, the Kernel-Managed Swap Facility (KMSF) allocates swap space for the default extended data segment of the process. For more information on this facility, see the *Kernel-Managed Swap Facility (KMSF) Manual*.

for TNS processes, this parameter is the address of a null-terminated OSS pathname of the Guardian swap file to be used for the default extended data segment of the process. The swap file must be an unstructured file. If the pathname is relative, it is resolved using the OSS current working directory.

For native processes, this parameter is ignored because TNS/R native processes do not need an extended swap file.

The default value is 0D.

For more information about the swap files, see Considerations for Resolving File Names on page 12-182 and DEFINE Considerations on page 12-142.

`Z^Priority`

is the initial execution priority to be assigned to the new process. Execution priority is a value in the range 1 through 199, where 199 is the highest possible priority.

If you specify the default value of -1, the priority of the caller is used. If you specify 0, a value less than -1, or a value greater than 199, error 2 is returned in ZSYS^DDL^PROCESSRESULTS.Z^TPCERROR.

`Z^CPU`

specifies the processor in which the new process is to run. If you specify the default value of -1, the caller's processor is used.

`Z^NameOptions`

specifies whether the process is to be named and, if so, whether the caller is supplying the name or the system must generate it.

This table summarizes the possible values for Z^NAMEOPTIONS. TAL literals are defined in the ZSYSTAL file. Literals in the zsysc file, for C programs, are the same as those for TAL except that they contain the underscore (_) character instead of the circumflex (^) character.

One of these values must be supplied:

Name (ZSYS^VAL^)	Value	Description
PCreatOpt^NamedBySys	2	Process is named; the system must generate a name. The generated name is four characters long, not including the /G/.
PCreatOpt^NamedBySys5	4	Process is named; the system must generate a name. The generated name is five characters long, not including the /G/.
PCreatOpt^NameInCall	1	Process is named; name is supplied in Z^ProcessName.
PCreatOpt^NoName	0	Process is not named; it can be named if the RUNNAMED program-file flag is set

If either the program file or the library file (if any) has the RUNNAMED program-file flag set, the system generates a name. The generated name is four characters long, not including the /G/, unless Z^NAMEOPTIONS is ZSYS^VAL^PCREATOPT^NAMEDBYSYS5. In which case, the name is five characters long, not including the /G/.

`Z^ProcessName`

is the address of a null-terminated string that specifies the name to be assigned to the new process. The name cannot include a node name. This parameter is relevant only when Z^NAMEOPTIONS has the value ZSYS^VAL^PCREATOPT^NAMEINCALL. For information about reserved process names, see Nowait Considerations on page 12-142 and Appendix B, Reserved Process Names.

For other values of Z^NAMEOPTIONS, this parameter should be set to the default value of 0D, because the system will generate a name.

`Z^HomeTerm`

is the address of a null-terminated string that specifies a file name designating the home terminal for the new process. If Z^HOMETERM is relative, it is resolved using the OSS current working directory.

Z^HOMETERM can be a terminal device or a named or unnamed user process. The default value of 0D indicates the home terminal of the caller. Note that the default home terminal of the caller can be a remote terminal.

`Z^MemoryPages`

For TNS processes, specifies the minimum number of memory pages allocated to the new process for user data. The actual amount of memory allocated is processor-dependent. If Z^MEMORYPAGES is set to either the default value of -1 or a value less than the value previously assigned by a compiler directive at compile time or by a Binder command at bind time, the previously assigned value is used. In any case, the maximum number of pages permitted is 64.

For native processes, this parameter is ignored. To specify the maximum size of the main stack, specify the `Z^MAINSTACKMAX` field. Alternatively, use the `nld` utility to set the TNS/R process attributes or the `eld` utility to set the TNS/E process attributes.

`Z^JobID`

is an integer (job ID) that specifies the job to be created. The new process is the first process of the job, and the caller is the job ancestor of the new process. This value is used by the NetBatch scheduler. For information about how to use this parameter, see Batch Processing Considerations on page 12-143.

The default value of -1 indicates that the new process is not a batch job.

`Z^CreateOptions`

provides information about the environment of the new process.

This table summarizes the possible values for Z^CREATEOPTIONS. More than one value can be specified. TAL literals are defined in the ZSYSTAL file. Literals

in the zsysc file, for C programs, are the same as those for TAL except that they contain the underscore (_) character instead of the circumflex (^) character.

Valid values for Z^CREATEOPTIONS are one or more of these:

Name (ZSYS^VAL^)	Value	Description
PCreatOpt^AllDefines	16	Propagate DEFINEs in Z^DEFINES and DEFINEs in the caller's context. In case of name conflicts, use the ones in Z^DEFINES. Otherwise, propagate DEFINEs as specified by other values.
PCreatOpt^AnyAncestor	64	If the caller is named, the process deletion message, if any, will go to whatever process has the calling process's name (regardless of sequence number) at that time.
PCreatOpt^Default	0	The default value, which is described with each of the other options.
PCreatOpt^DefEnabled	2	See PCREATOPT^DEFOVERRIDE.
PCreatOpt^DefineList	8	Propagate DEFINEs in Z^DEFINES only. Otherwise, propagate only the DEFINEs in the caller's context.
PCreatOpt^DefOverride	4	Enable DEFINEs if PCREATOPT^DEFENABLED is specified. Disable DEFINEs if PCREATOPT^DEFENABLED is not specified. Otherwise, use caller's DEFINE mode.
PCreatOpt^FrcLowOver	32	Ignore the value of the caller's inherited force-low pin attribute. Otherwise, use the value of the caller's inherited force-low pin attribute.
PCreatOpt^LowPin	1	Require low PIN (in range 0 through 254). Otherwise, assign any PIN.

If you specify ZSYS^VAL^PCREATOPT^LOWPIN, the program is run at a low PIN. If you do not specify ZSYS^VAL^PCREATOPT^LOWPIN, the program runs at a PIN of 256 or higher if its program file and library file (if any) have the HIGHPIN program-file flag set and if a high PIN is available. However, if the calling process has the inherited force-low attribute set, the new process is forced into a low PIN even if all the other conditions for running at a high PIN are met. For more information, see Compatibility Considerations on page 12-46.

For more information on DEFINEs, see DEFINE Considerations on page 12-142.

`Z^Defines`

is the address of a null-terminated string that specifies a set of DEFINEs to be propagated to the new process. The value of Z^DEFINES must be exactly Z^DEFINESLEN bytes long. The set of DEFINEs should have been created through one or more calls to the DEFINESAVE procedure. DEFINEs are propagated according to the values specified in Z^CREATEOPTIONS. For details, see DEFINE Considerations on page 12-142.

The default value is 0D.

`Z^DefinesLen`

specifies the length of the Z^DEFINES field.

The default value is 0.

`Z^DebugOptions`

sets the debugging attributes for the new process.

This table summarizes the possible values for Z^DEBUGOPTIONS. TAL literals are defined in the ZSYSTAL file. Literals in the zsysc file, for C programs, are the same as those for TAL except that they contain the underscore (_) character instead of the circumflex (^) character.

Valid values for Z^DebugOptions are as follows:

Name (ZSYS^VAL^)	Value	Description
PCreatOpt^DbgOverrid e	2	Use debugger and saveabend options specified regardless of program-file flag settings. Otherwise, use the program-file flag settings. Debugger and saveabend options are specified by ZSYS^VAL^PCREATOPT^SAVEABEND and ZSYS^VAL^PCREATOPT^RUND described in this table.
PCreatOpt^Default	0	The debugger and saveabend default values are set from the flags in the program file (set either by compiler directives at compile time, `nld` flag at link time, or Binder command at bind time) after these options are ORed with the corresponding states of the calling process.
PCreatOpt^INSPECT	1	Use the Inspect debugger. Otherwise, use the debugger set by the program-file flag settings.
PCreatOpt^RUND	8	Enter Debug or the Inspect debugger at the first executable instruction of the program's MAIN procedure. If this option is not selected, begin normal program execution.
PCreatOpt^SaveAbend	4	If the process traps, create a saveabend file and use the Inspect debugger (regardless of whether ZSYS^VAL^PCREATOPT^INSPECT is selected). If this option is not selected and the process traps, do not create a saveabend file.

`Z^PFSSize`

for RVUs before G06.00, specifies the size in bytes of the process file segment (PFS) of the new process. In G-series RVUs, maximum PFS size is 8 MB. In H-series RVUs, maximum PFS size is 32 MB. A value in this range overrides the `nld` or Binder value stored in the program file.

If you specify 0, the `nld` or Binder value is used. If the `nld` or Binder value is to be used and it is 0, a default value of 262,144 bytes (256 KB or 2 segments) is used in most cases. For SQL programs, a default value of 393,216 bytes (384 KB or 3 segments) is used instead.

In the G06.00 and subsequent G-series RVUs, you do not need to specify the PFS size of a new process. If you do, the specification is ignored. Instead, all processes have PFS size that can grow up to 8 MB.

Z^OSSOptions

The valid value for Z^OSSOPTIONS is as follows:

Name (ZSYS^VAL^)	Value	Description
PSpawnOpt^OSSDefault	0	The default value is the only value available.

Z^MainStackMax

specifies the maximum size, in bytes, of the process main stack. The specified size cannot exceed 32 MB.

The default value of 0D indicates that the main stack can grow to 1MB. For most processes, the default value is adequate.

Z^HeapMax

for native processes only, specifies the maximum size, in bytes, of the process heap. Note that the sum of the size of the heap and the size of global data cannot exceed 384 MB.

The default value of 0D indicates that the heap can grow to the default value of 16 MB. The initial heap size of a process is zero bytes. For most processes, the default value is adequate.

Z^SpaceGuarantee

specifies the minimum size, in bytes, of the amount of space that the process reserves with the Kernel-Managed Swap Facility for swapping. For more information on this facility, see the *Kernel-Managed Swap Facility (KMSF) Manual*. The value provided is rounded up to a page size boundary of the processor. If the requested amount of space is not available, PROCESS_SPAWN_ returns error 55.

When the default value of 0D is used, the amount of space reserved is determined by the value specified in the object file for a native process or by the operating system for a TNS or accelerated process.

Structure Definition for *process-results*

The *process-results* parameter provides Guardian information on the outcome of the PROCESS_SPAWN_ procedure call.

In the TAL ZSYSTAL file, the structure for the *process-results* parameter is defined as:

```
STRUCT ZSYS^DDL^PROCESSRESULTS^DEF (*);
   BEGIN
   INT(32) Z^LEN;
   STRUCT Z^PHANDLE;
      BEGIN
      STRUCT Z^DATA;
         BEGIN
         STRING ZTYPE;
         FILLER 19;
         END;
      INT Z^WORD[0:9] = Z^DATA;
      STRUCT Z^BYTE = Z^DATA;
         BEGIN STRING BYTE [0:19]; END;
      END;
   INT(32) Z^PID;
   INT(32) Z^ERRNO;
   INT     Z^TPCERROR;
   INT     Z^TPCDETAIL;
   END;
```

For TAL programs, this value must be specified:

Field Name	Default Value
Z^LEN	$OFFSET(ZSYS^DDL^PROCESSRESULTS.Z^TPCDETAIL) + $LEN(ZSYS^DDL^PROCESSEXTENSION.Z^TPCDETAIL)

In the C tdmext.h header file, the structure for the *process-results* parameter is defined as:

```
typedef struct process_extension_results {
   long    pr_len;
   short   pr_phandle[10];
   long    pr_pid;
   long    pr_errno;
   short   pr_TPCerror;
   short   pr_TPCdetail;
} process_extension_results_def;
```

The **tdmext.h** header file is not kept current when new error codes are defined for process creation functions. The list of **_TPC_** macros described in this reference page is not complete; for a current description of error macros and error codes, see the Guardian header file $SYSTEM.ZSPIDEF.ZGRDC or Table 12-3, Summary of Process Creation Errors in the *Guardian Procedure Calls Refrence Manual*.

C programs should initialize the process_extension_results structure by using the #define DEFAULT_PROCESS_EXTENSION_RESULTS in the tdmext.h header file.

Z^Len

is the length of the ZSYS^DDL^PROCESSRESULTS structure. Because the structure is subject to change, Z^LEN is used by PROCESS_SPAWN_ to identify the version of the structure.

Z^Phandle

returns the process handle of the new process. If you created the process in a nowait manner, the process handle is returned in the completion message sent to $RECEIVE rather than in this parameter.

Z^PID

returns the process ID of the new process. If you created the process in a nowait manner, then the returned value is 0D and the process ID is returned in the completion message sent to $RECEIVE. If an error occurs (Z^ERRNO or Z^TPCERROR are not 0), then the returned value is -1D.

Z^Errno

indicates the outcome of the process creation.

The more common OSS errno values returned in Z^ERRNO are:

Z^ERRNO	Description
0	No error. The corresponding OSS errno value is ENOERR.
4002	No such pathname exists. The corresponding OSS errno value is ENOENT.
4005	A physical input or output error occurred. The corresponding OSS errno value is EIO.
4007	The argument list, specified by the *argv* parameter, is too long. The corresponding OSS errno value is E2BIG.
4008	The *oss-program-file* parameter has the appropriate permissions, but is not in the format for executable files. The corresponding OSS errno value is ENOEXEC.
4009	A file descriptor specified in the *fdinfo* parameter is either out of range or does not exist. The corresponding OSS errno value is EBADF.
4011	System resources are inadequate. The corresponding OSS errno value is EAGAIN.
4012	There is insufficient user memory to create the process. The corresponding OSS errno value is ENOMEM.
4013	Search permission is denied on a component of the pathname prefix. The corresponding OSS errno value is EACCES.
4014	A specified parameter has an invalid address. The corresponding OSS errno value is EFAULT.

Z^ERRNO	Description
4020	A prefix within a pathname refers to a file other than a directory. The corresponding OSS errno value is ENOTDIR.
4022	Either a parameter in the parameter list is invalid or a required parameter is omitted. The corresponding OSS errno value is EINVAL.
4126	Operation timed out. The timeout value was reached before a binary semaphore could be locked. The corresponding OSS errno value is ETIMEDOUT.
4131	The pathname or a component of the pathname is longer than PATH_MAX characters. (PATH_MAX is a symbolic constant that is defined in the OSS limits.h header file.) See Appendix D, File Names and Process Identifiers, for pathname syntax. The corresponding OSS errno value is ENAMETOOLONG.
4203	OSS is not running or is not installed. The corresponding OSS errno value is EOSSNOTRUNNING.
4212	An error occurred during the invocation of a Guardian DEFINE. The corresponding OSS errno value is EDEFINEERR.

Z^TPCError

indicates the outcome of the Guardian process creation. This parameter is the same as the *error* parameter reported by PROCESS_LAUNCH_. For details, see Table 12-3 on page 12-111 and Table 12-4 on page 12-120. See also Nowait Considerations on page 12-46.

Z^TPCDetail

returns additional information about some classes of Guardian errors. This parameter is the same as the *error-detail* parameter reported by PROCESS_LAUNCH_. For details, see Table 12-3 on page 12-111 and Table 12-4 on page 12-120.

Nowait Considerations

If you call this procedure in a nowait manner, the results are returned in the nowait PROCESS_SPAWN_ completion message (-141), not the output parameters of the procedure. The format of this completion message is described in the *Kernel-Managed Swap Facility (KMSF) Manual*. If Z^TPCError is not 0, no completion message is sent to $RECEIVE. Errors can be reported either on return from the procedure, in which case the error output parameters might be meaningful, or through the completion message sent to $RECEIVE.

Considerations for Resolving File Names

- All file names are specified using OSS pathname syntax and are resolved using the caller's OSS current working directory.
- For TNS and accelerated processes on RVUs preceding the D42 RVU, if the Z^SWAPFILENAME or Z^EXTSWAPFILENAME field of the *process-extension* parameter is specified and
  - A file with that name exists, that file is used for memory swaps of the user data stack (Z^SWAPFILENAME) or of the default extended data segment (Z^EXTSWAPFILENAME) during execution of the process; if no file of that name exists, then a file of that name and of the necessary size is created and used for swapping.
  - Contains the name of a temporary file that is already in use, an error is returned.
  - Contains only the disk volume name, then a temporary file is created on the specified disk device.
  - Is equal to 0D, then the SWAP volume name in the =_DEFAULTS DEFINE is used if available. Otherwise, the system chooses where to place the file.
- Resolving the problem of spawning remote shell scripts
  - Use PROCESS_SPAWN_ to spawn a remote shell and pass the name of the script as one of its arguments. The shell will run the script.
  - Spawn a local shell and use the Expand file system to read the remote shell script.

Considerations for Resolving External References

- Program file and user library file differences

  A user library is an program file containing one or more procedures. The difference between a program file and the library file is that the library file cannot contain a MAIN procedure but a program file must contain a MAIN procedure. Undefined external references in a program file are resolved from the user library, if any, or the system library. Unresolved references in a library are resolved only from the system library.
- Library conflict PROCESS_SPAWN_ error

  The library file for a process can be shared by any number of processes. However, when a program is shared by two or more processes, all processes must have the same user library configuration; that is, all processes sharing the program either have the same user library or have no user library. A library conflict error occurs when there is already a copy of the program running with a library configuration different from that specified in the call to PROCESS_SPAWN_.
- I/O error to the home terminal

An I/O error to the home terminal can occur if there are undefined externals in the program file and PROCESS_SPAWN_ is unable to open or write to the home terminal to display the undefined externals messages. The Z^TPCDETAIL field of the *process-results* parameter contains the file-system error number that resulted from the open or write that failed.

Considerations for Reserved Names

The operating system reserved process name space includes these names: /G/X*name*, /G/Y*name*, /G/Z*name*, where *name* is from 1 through 4 alphanumeric character. You should not use names of these forms in any application. System-generated process names (from PROCESS_SPAWN_, PROCESS_CREATE_, NEWPROCESS[NOWAIT], PROCESSNAME_CREATE_ , CREATEPROCESSNAME, and CREATEREMOTENAME) are selected from this set of names. For more information about reserved process names, see Appendix B, Reserved Process Names.

Keeping Track of OSS Child Processes

Because OSS child processes can migrate from one processor to another, the caller process of an OSS process should monitor all processors to determine whether its child process is still alive if a processor goes down. these two examples show how a caller process should handle a processor down message:

- A child process migrates from a processor that is about fail to a running processor:
  1. The child process migrates from processor 5 to a new process handle on processor 7 by calling one of the OSS `tdm_exec` set of functions.
  2. processor 5 fails.
  3. The caller process receives the processor down message from processor 5. At this point, the caller process does not know whether its OSS child process still exists, because the child process could have migrated to another processor before the failure in processor 5. The caller process calls PROCESS_GETINFOLIST_ with the OSS process ID of the child process and obtains the new process handle of the OSS process indicating that it still exists.
  4. The caller process receives the process deletion message with a -12 completion code (indicating that the child process has migrated to a new process handle by calling one of the OSS `tdm_exec` set of functions).
- A child process migrates from a processor that is running to a processor that fails:
  1. The child process migrates from processor 2 to a new process handle on processor 6 by calling one of the OSS `tdm_exec` set of functions.
  2. processor 6 fails.

3. The caller process receives the processor down message from processor 6. At this point, the caller process does not know whether its OSS child process still exists, because the child process could have migrated from processor 2 to processor 6. The caller process calls PROCESS_GETINFOLIST_ with the OSS process ID of the child process. PROCESS_GETINFOLIST_ returns error 4 indicating that the specified process does not exist.
4. The caller process receives the process deletion message with a -12 completion code (indicating that the child process has migrated to a new process handle by calling one of the OSS `tdm_exec` set of functions), but the child process no longer exists.

Creator Access ID and Process Access ID

The creator access ID (CAID) of the new process is always the same as the process access ID (PAID) of the creator process. The process access ID of the new process is the same as that of the creator process unless the program file has the PROGID attribute set; in that case, the process access ID of the new process is the same as the NonStop opeating system user ID of the program file's owner, and the new process is always local.

Compatibility Considerations

- If the new process is unnamed, it must be run at a low PIN if it is to be accessible to processes which cannot access high-PIN processes.
- If the new process has a high PIN and also has a name with up to five characters (not counting the /G/), it is accessible to any process running on the same system.
- For further information on compatibility, see the *Guardian Programmer's Guide* and the *Guardian Application Conversion Guide*.
- If a client attempts a nontrithroughl call to the OSS `chroot()` function, the client cannot create remote processes, because `/E` will not be visible.

DEFINE Considerations

- DEFINEs are propagated to the new process from either the process context of the caller, from a caller-supplied buffer containing DEFINEs collected by calls to the DEFINESAVE procedure, or from both of these. DEFINEs are propagated to the new process according to the DEFINE mode of the new process and the propagation option specified in the Z^CREATEOPTIONS field of the *process_extension* parameter. If both sets of DEFINEs are propagated and both sets contain a DEFINE with the same name, the DEFINE in the caller-supplied buffer is used. When a caller is creating its backup, the caller's DEFINEs are always propagated, regardless of the options chosen.

  The =_DEFAULTS DEFINE is always propagated, regardless of the options chosen. If the DEFINE buffer contains a =_DEFAULTS DEFINE, that one is

propagated; otherwise, the =_DEFAULTS DEFINE in the caller's context is propagated.

Buffer space for DEFINEs being propagated to a new process is limited to 2 MB whether the process is local or remote. However, the caller can propagate only as many DEFINEs as the child's PFS can accommodate in the buffer space for the DEFINEs themselves and in the operational buffer space needed to do the propagation. The maximum number of DEFINEs that can be propagated varies depending upon the size of the DEFINEs being passed.

- When a process is created, its DEFINE working set is initialized with the default attributes of CLASS MAP.
- For TNS processes, the Z^SWAPFILENAME and Z^EXTSWAPFILENAME fields of the *process_extension* parameter can be DEFINE names; PROCESS_SPAWN_ uses the disk volume or file given in the DEFINE. If either Z^SWAPFILENAME or Z^EXTSWAPFILENAME contains a DEFINE name but no such DEFINE exists, the procedure behaves as if no name were specified. This feature of accepting names of nonexistent DEFINEs as input gives the programmer a convenient mechanism that allows, but does not require, user specification of the location of the swap file or extended swap file.
- For each process, a count is kept of the changes to that process's DEFINEs. This count is always 0 for newly created processes. The count is incremented each time the procedures DEFINEADD, DEFINEDELETE, DEFINESETMODE, and DEFINEDELETEALL are invoked and a consequent change to the process context occurs. In the case of DEFINEDELETE and DEFINEDELETEALL, the count is incremented by 1 even if more than one DEFINE is deleted. The count is also incremented if the DEFINE mode of the process is changed. If a call to the CHECKDEFINE procedure causes a DEFINE in the backup to be altered, deleted, or added, then the count for the backup process is incremented.

Batch Processing Considerations

Note. The job ancestor facility is intended for use by the NetBatch product. Other applications that use this facility might be incompatible with the NetBatch product.

- When the process being created is part of a batch job, PROCESS_SPAWN_ sends a job process creation message to the job ancestor of the batch job. (See the discussion of "job ancestor" in the *Guardian Programmer's Guide*.) The message identifies the new process and contains the job ID as originally assigned by the job ancestor. This enables the job ancestor to keep track of all the processes belonging to a given job.

  For the format of the job process creation message, see the *Guardian Procedure Errors and Messages Manual*.
- PROCESS_SPAWN_ can create a new process and establish that process as a member of the caller's batch job. In that case, the caller's job ID is propagated to the new process. If the caller is part of a batch job, then to start a new process that

is part of the caller's batch job, set the Z^JOBID field of the *process-extension* parameter to -1.

- PROCESS_SPAWN_ can create a new process separate from any batch job, even if the caller is a process that belongs to a batch job. In that case the job ID of the new process is 0. To start a new process that is not part of a batch job, specify 0 for Z^JOBID.
- PROCESS_SPAWN_ can create a new batch job and establish the new process as a member of the newly created batch job. In that case, the caller becomes the job ancestor of the new job; the job ID supplied by the caller becomes the job ID of the new process. To start a new batch job, specify a nonzero value (other than -1) for the Z^JOBID field of the *process-extension* parameter.

  A job ancestor must not have a process name that is longer than four characters (not counting the dollar sign). When the caller of PROCESS_SPAWN_ is to become a job ancestor, it must conform to this requirement.
- When the Z^JOBID field of the *process-extension* parameter is set to -1:
  - If the caller is not part of a batch job, then neither is the newly created process; its job ID is 0.
  - If the caller is part of a batch job, then the newly created process is part of the same job because its job ID is propagated to the new process.
- Once a process belongs to a batch job, it remains part of the job.

Safeguard Considerations

For information on processes protected by the Safeguard product, see the *Safeguard Reference Manual*.

Related Programming Manuals

For programming information on batch processing, see the appropriate NetBatch manual. For programming information on Open System Services and PROCESS_SPAWN_ programming examples, see the *Open System Services Programmer's Guide*.

PROCESS_STOP_ Procedure

Summary

The PROCESS_STOP_ procedure deletes a process or process pair. When this procedure is used to delete a Guardian process or an OSS process, a process deletion system message is sent to the mom of the process and to any other process that is entitled to receive the message. When this procedure is used to delete an OSS process, a SIGCHLD signal and the OSS process termination status are sent to the OSS caller.

A process can use PROCESS_STOP_ to:

- Delete itself
- Delete its backup
- Delete another process

Syntax for C Programmers

```
#include <cextdecs(PROCESS_STOP_)>

short PROCESS_STOP_ ( [ short *processhandle ]
                     ,[ short specifier ]
                     ,[ short options ]
                     ,[ short completion-code ]
                     ,[ short termination-info ]
                     ,[ short *spi-ssid ]
                     ,[ const char *text ]
                     ,[ short length ] );
```

The parameter *length* specifies the length in bytes of the character string pointed to by *text*. The parameters *text* and *length* must either both be supplied or both be absent.

Syntax for TAL Programmers

```
error := PROCESS_STOP_ [ ( [ processhandle ]          ! i
                         ,[ specifier ]               ! i
                         ,[ options ]                 ! i
                         ,[ completion-code ]         ! i
                         ,[ termination-info ]        ! i
                         ,[ spi-ssid ]                ! i
                         ,[ text:length ] ) ];        ! i:i
```

Parameters

error returned value

INT

unless the caller successfully stops itself, returns a file-system error number that indicates the outcome of the operation. See “Considerations” for information about interpreting the error numbers that are returned.

processhandle input

INT .EXT:ref:10

specifies the process handle of the process to be stopped. If this parameter is omitted or null, the caller is stopped. The null process handle is one which has -1 in each word (Refer to Guardian procedure call, PROCESSHANDLE_NULLIT_). However, PROCESS_STOP also treats a process handle with -1 in the first word as a null process handle.

specifier input

INT:value

for a named process pair, indicates whether both members should be stopped. Valid values are:

0 Stop the specified process only.
1 Stop both members of current instance of named process pair if the specified process is part of a named process pair; otherwise stop the specified process.
2 Stop the caller’s opposite member, but not the caller, if it is part of a named process pair. *processhandle* is ignored.

The default is 0.

If *processhandle* is null or omitted, a *specifier* value of 0 constitutes a request to stop the caller and a value of 1 constitutes a request to stop the caller’s process pair (if the caller is a member of a process pair).

options input

INT:value

specifies whether the process is being stopped because of a normal or abnormal condition. Valid values are:

<0:14>		Reserved (specify 0)
<15>	0	Normal termination (STOP)
	1	Abnormal termination (ABEND)

The default is 0.

These parameters supply completion-code information, which consists of four items: the completion code, a numeric field for additional termination information, a subsystem identifier in SPI format, and an ASCII text string. These items apply only when the caller is terminating itself.

completion-code input

INT:value

is the completion code to be returned in the process deletion system message and, for a terminating OSS process, in the OSS process termination status. Specify this parameter only if the calling process is terminating itself and you want to return a completion code value other than the default value of 0 (STOP) or 5 (ABEND).

A nonprivileged caller cannot pass a negative value for *completion-code*.

For a list of completion codes, see Appendix C, Completion Codes.

termination-info input

INT:value

specifies the Subsystem Programmatic Interface (SPI) error number that identifies what caused the process to stop itself. For more information on SPI error numbers and subsystem IDs, see the *SPI Programming Manual*. If *termination-info* is not specified, the default is 0.

If *termination-info* is specified, *spi-ssid* and *text*:*length* should be supplied.

spi-ssid input

INT .EXT:ref:6

is a subsystem ID (SSID) that identifies the subsystem defining the *termination-info*. The format and use of the SSID is described in the *SPI Programming Manual*.

text:*length* input:input

STRING .EXT:ref:*, INT:value

if present and *length* is not 0, is a string of ASCII text to be sent as part of the process deletion system message. If used, the value of *text* must be exactly *length* bytes long. The maximum length is 80 bytes.

Considerations1

- When PROCESS_STOP_ executes, all open files associated with the deleted process are closed. If a process had BREAK enabled, BREAK is disabled.
- Recipients of process deletion system messages

  When a process is stopped, these processes receive a process deletion system message:

  - The mom of the stopped process (if any)
  - The ancestor of the stopped process if the stopped process is a single named process or part of a named process pair where both members of the pair are stopped (only one message is received when both members of a named process pair are stopped)
  - The job ancestor (GMOM) of the stopped process if the stopped process is part of a batch job

  If the caller of PROCESS_STOP_ is also the mom, ancestor, or job ancestor of the process being terminated, it receives a process deletion system message.
- Recipients of OSS process termination status

  If the stopped process was an OSS process, then its OSS caller process receives a SIGCHLD signal and the OSS process termination status.

  See the `wait(2)` function reference pages either online or in the *Open System Services System Calls Reference Manual* for details on interpreting the OSS process termination status.
- Differences between ABEND and STOP options

  When used to stop the calling process, the ABEND and STOP options (*options*.<15>) operate almost identically; they differ in the system messages that are sent and the default completion codes that are reported. In addition, PROCESS_STOP_ with the ABEND option specified causes a saveabend file to be created if the process's SAVEABEND attribute is set to ON. See the *Inspect Manual* for information about saveabend files.

  For the exact formats of the process deletion system messages, see the *Guardian Procedure Errors and Messages Manual*. Note that PROCESS_STOP_ can send either a C-format message or a D-format message, depending on the recipient. (A process can specify, when it opens $RECEIVE, that it wants to receive either C-format messages or D-format messages. For details, see FILE_OPEN_ Procedure.)

PROCESS_STOP_ sends a default completion code of 0 when the STOP option is specified; it sends a completion code of 5 when the ABEND option is specified.

- Rules for stopping a Guardian process: process access IDs and creator access IDs

  If the process is a local process and the request to stop it is also from a local process, these user IDs or associated processes can stop the process:

  - local super ID
  - the process's creator access ID (CAID) or the group manager of the CAID
  - the process's process access ID (PAID) or the group manager of the PAID

  If the process is a local process, a remote process cannot stop it.

  If the process is a remote process running on the local system and the request to stop it is from a local process, these user IDs or associated processes can stop the process:

  - local super ID
  - the process's creator access ID (CAID) or the group manager of the CAID
  - the process's process access ID (PAID) or the group manager of the PAID

  If the process is a remote process on the local system and the request to stop it is from a remote process, these user IDs or associated processes can stop the process:

  - a network super ID
  - the process's network process access ID
  - the process's network process access ID group manager
  - the process's network creator access ID
  - the process's network creator access ID group manager

  Being local on a system means either that the process has logged on by successfully calling USER_AUTHENTICATE_ (or VERIFYUSER) on the system or that the process was created by a process that had done so. A process is also considered local if it is run from a program file that has the PROGID attribute set.

- Rules for stopping an OSS process

  The same rules apply when stopping an OSS process with the PROCESS_STOP_ procedure as apply for the OSS `kill()` function. See the `kill(2)` function reference page either online or in the *Open System Services System Calls Reference Manual*.

- Rules for stopping any process: stop mode

When a process tries to stop another process, another item checked is the stop mode of that process. The stop mode is a value associated with every process that determines what other processes can stop it. The stop mode, set by the SETSTOP procedure, is defined as follows:

0 Any other process can stop the process.
1 Only the process qualified by the above rules can stop the process.
2 No other process can stop the process.

The process can always stop itself.

- Errors other than 0 can be returned by PROCESS_STOP_ under these conditions:
  - If the process (or process pair) does not exist, error 11 is returned.
  - If the stop request passes the security checks but the process is running at stop mode 2, the stop request is queued pending the reduction of the stop mode to 1. Error 638 is returned.
  - If the stop request does not pass the security checks and the process is running at stop mode 1 or 2, the stop request is queued pending the reduction of the stop mode to 0. Error 639 is returned.
  - If it is not possible to communicate with the processor where the process is running, error 201 is returned.
- Returning control to the caller before the process is stopped

  When *error* is 0, 638, or 639, PROCESS_STOP_ returns control to the caller before the specified process is actually stopped. If *error* is 0, the process does not execute any more user code. However, you should make sure that the process has terminated before you attempt to access a file that the process had open with exclusive access or before you try to create a new process with the same name. The best way to be sure that a process has terminated is to wait for the process deletion message.
- Stopping a process that has the Inspect or saveabend attribute set

  If the process being stopped has either the Inspect attribute or the saveabend attribute set, and if DMON exists, PROCESS_STOP_ returns error 0 but deletion of the process is delayed until DMON approves it. In the case of an abnormal termination (ABEND), DMON creates a saveabend file if the saveabend attribute is set.
- In response to the PROCESS_STOP_ procedure, the operating system supplies a completion code in the process deletion message and, for OSS processes, in the OSS process termination status as follows:
  - If a process calls PROCESS_STOP_ on another process, the system supplies a completion code value of 6.
  - If a process calls PROCESS_STOP_ with the STOP option on itself but does not supply a completion code, the system supplies a completion code value of 0.

- If a process calls PROCESS_STOP_ with the ABEND option on itself but does not supply a completion code, the system supplies a completion code value of 5.

For a list of completion codes, see Appendix C, Completion Codes.

- If PROCESS_STOP_ is issued by the backup process of a process pair, with a *specifier* parameter value of 1, the intent is to stop the primary process and itself. However, if the primary process is running in stop-mode 2, then only the backup process is stopped because the primary process is running in stop-mode 2. If the primary process continues to run in stop-mode 2 and tries to re-create the backup process, process-creation error 11,45 is returned. This error also occurs when a primary process issues PROCESS_STOP_ with the *specifier* parameter set to 1 to stop an unstoppable backup process and itself.

NetBatch Considerations

- The PROCESS_STOP_ procedure supports NetBatch by:
  - returning the completion code information in the process deletion system message
  - returning the process processor time in the process deletion system message
  - sending a process deletion system message to the job ancestor (GMOM) of the job, as well as to the mom and ancestor of the process, when any process in the job is terminated

Safeguard Considerations

For information on processes protected by Safeguard, see the *Safeguard Reference Manual*.

OSS Considerations

- When an OSS process is stopped by the PROCESS_STOP_ procedure, either by calling the procedure to stop itself or when some other process calls the procedure, the OSS caller process receives a SIGCHLD signal and the OSS process termination status. See the `wait(2)` function reference page either online or in the *Open System Services System Calls Reference Manual* for details on the OSS process termination status.

  In addition, a process deletion system message is sent to the MOM, GMOM, or ancestor process according to the usual Guardian rules. The OSS process ID of the terminated process is included in the process deletion message.

- When the PROCESS_STOP_ procedure is used to stop an OSS process other than the caller, the process handle must be specified in the call. The effect is the same as if the OSS `kill()` function was called with the input parameters as follows:

- The *signal* parameter set to SIGKILL to stop the process or SIGABEND to abend the process
- The *pid* parameter set to the OSS process ID of the process identified by *processhandle* in the PROCESS_STOP_ call

- The security rules that apply to stopping an OSS process using PROCESS_STOP_ are the same as those that apply to the OSS `kill()` function. See the `kill(2)` function reference page either online or in the *Open System Services System Calls Reference Manual* for details.

Examples

```
INT stop^option;
    .
    .
error := PROCESS_STOP_ ( proc^handle ); ! stop the identified
                                        !  process (normal
                                        !   termination)
stop^option := 1;                         ! set ABEND flag
error := PROCESS_STOP_ ( , , stop^option );  ! stop self
                                             ! (abnormal
                                             !  termination)
```

Related Programming Manual

For programming information about the PROCESS_STOP_ procedure, see the *Guardian Programmer's Guide*. For information on batch processing, see the appropriate NetBatch manual.

PROCESS_SUSPEND_ Procedure

Summary

The PROCESS_SUSPEND_ procedure places a process or process pair into the suspended state, preventing that process from being active (that is, from executing instructions). A process can also be suspended by a call to the SUSPENDPROCESS procedure, or by a TACL SUSPEND command. The process or process pair can be reactivated by a subsequent call to PROCESS_ACTIVATE_ or ACTIVATEPROCESS or by a TACL ACTIVATE command.

Syntax for C Programmers

```
#include <cextdecs(PROCESS_SUSPEND_)>

short PROCESS_SUSPEND_ ( short *processhandle
                         ,[ short specifier ] );
```

Syntax for TAL Programmers

```
error := PROCESS_SUSPEND_ ( processhandle          ! i
                           ,[ specifier ] );       ! i
```

Parameters

error returned value

INT

indicates the outcome of the operation. It returns one of these file-system errors:

0	Process successfully suspended.
2	Process is already in the suspended state.
11	Process does not exist.
48	Security violation.
201	Unable to communicate with processor where the process is running.

processhandle input

INT .EXT:ref:10

specifies the process handle of the process to be suspended.

specifier input

INT:value

for a named process pair, indicates whether both members should be suspended. Valid values are:

0 Suspend the specified process only.

1 Suspend both members of current instance of named process pair if the specified process is part of a named process pair; otherwise suspend the specified process.

The default is 0.

Considerations

- Reactivating a process

  You can reactivate a suspended process or process pair by calling PROCESS_ACTIVATE_. You can also reactivate it by calling ACTIVATEPROCESS, but you must have a process ID to identify the process. A process handle can be converted to a process ID by a call to PROCESSHANDLE_TO_CRTPID_ , but the conversion will fail if the PIN of the process is greater than 255.

- Security

  When PROCESS_SUSPEND_ is called on a Guardian process, the caller must be the super ID, the group manager of the process access ID, or a process with the same process access ID as the process or process pair being suspended. For information about the process access ID, see General Considerations on page 12-62 and the *Guardian User's Guide*.

  The caller must be local to the same system as the specified process. A process is considered to be local to the system on which its creator is local. A process is considered to be remote, even if it is running on the local system, if its creator is remote. (In the same manner, a process running on the local system whose creator is also running on the local system might still be considered remote because it's creator's creator is remote.)

  A remote process running on the local system can become a local process by successfully logging on to the local system using a call to the USER_AUTHENTICATE_ (or VERIFYUSER) procedure. After a process logs on to the local system, any processes that it creates are considered local.

  When PROCESS_SUSPEND_ is called on an OSS process, the security rules that apply are the same as those that apply when calling the OSS `kill()` function.

See the `kill(2)` function reference page either online or in the *Open System Services System Calls Reference Manual* for details.

Safeguard Considerations

For information on processes protected by Safeguard, see the *Safeguard Reference Manual*.

OSS Considerations

When used on an OSS process, PROCESS_SUSPEND_ has the same effect as calling the OSS `kill()` function with the input parameters as follows:

- The *`signal`* parameter set to SIGSTOP
- The *`pid`* parameter set to the OSS process ID of the process identified by *`processhandle`* in the PROCESS_SUSPEND_ call

The SIGSTOP signal is delivered to the target process. The SIGCHLD signal is delivered to the caller of the target process.

Example

```
error := PROCESS_SUSPEND_ ( proc^handle );
```

Related Programming Manual

For programming information about the PROCESS_SUSPEND_ procedure, see the *Guardian Programmer's Guide*.

PROCESS_WAIT_

Summary

The PROCESS_WAIT_ procedure takes a 32-bit mask value and a timeout value as parameters, and returns a 32-bit mask value.

PROCESSACCESSID Procedure (Superseded by PROCESS_GETINFOLIST_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The PROCESSACCESSID procedure is used to obtain the process access ID (PAID) of the calling process.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
access-id := PROCESSACCESSID;
```

Parameters

access-id returned value

INT

returns the process access ID (PAID) of the caller in this form:

`<0:7>` group number

`<8:15>` member number

Considerations

- Process access ID (PAID) compared to creator access ID (CAID)

  For a given process, an access ID is a word in the process control block (PCB) that contains a group number in the left byte and a member number in the right byte. There are two access IDs used in the operating system.

The process access ID (PAID) is returned from the PROCESSACCESSID procedure and is normally used for security checks when a process attempts to access a disk file.

The creator access ID (CAID) is returned from the CREATORACCESSID and identifies the user who created the process. It is normally used, often with the PAID, for security checks on interprocess operations such as stopping a process, creating a backup for a process, and so on.

The PAID and the CAID usually differ only when a process is run from a program file that has the PROGID attribute set. This attribute is usually set with the File Utility Program (FUP) SECURE command and PROGID option. In such a case, the process access ID returned by PROCESSACCESSID is the same as the user ID of the program file's owner.

Both the PAID and the CAID are returned from the PROCESS_GETINFO[LIST]_ procedures. See the *Guardian User's Guide* for information about process access IDs.

PROCESSFILESECURITY Procedure (Superseded by PROCESS_SETINFO_ Procedure or PROCESS_GETINFOLIST_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The PROCESSFILESECURITY procedure is used to examine or set the file security for the current process. This is the security used for any file creation attempts following a call to PROCESSFILESECURITY.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
old-security := PROCESSFILESECURITY ( [ security ] );    ! i
```

Parameters

old-security returned value

INT

is the old file security.

security input

INT:value

is the new file security. The security bits are:

```
<0:3>    0
<4:6>    ID code allowed for read
<7:9>    ID code allowed for write
<10:12>  ID code allowed for execute
<13:15>  ID code allowed for purge
```

ID code can be one of these:

0 Any user (local)
1 Member of owner's group (local)
2 Owner (local)
4 Any user (local or remote)
5 Member of owner's community (local or remote)
6 Owner (local or remote)
7 Super ID only (local)

If *security* is omitted, PROCESSFILESECURITY returns the current security information in *old-security* without changing it.

Example

```
OLD^SECURITY := PROCESSFILESECURITY ( SECURITY );
```

PROCESSHANDLE_COMPARE_ Procedure

Summary

The PROCESSHANDLE_COMPARE_ procedure compares two process handles and reports whether they are identical, represent different processes of the same process pair, or different.

PROCESSHANDLE_COMPARE_ is primarily useful for determining whether processes form a process pair. You can determine whether two process handles are identical by doing a ten-word unsigned comparison.

Syntax for C Programmers

```
#include <cextdecs(PROCESSHANDLE_COMPARE_)>

short PROCESSHANDLE_COMPARE_ ( short *processhandle-1
                              ,short *processhandle-2 );
```

Syntax for TAL Programmers

```
status := PROCESSHANDLE_COMPARE_ ( processhandle-1     ! i
                                  ,processhandle-2 );  ! i
```

Parameters

status returned value

INT

returns the result of the comparison. Valid values are:

0 Process handles are unrelated.
1 Process handles are not identical but designate a process pair.
2 Process handles are identical.

processhandle-1 input

INT .EXT:ref:10

is one of the process handles to be compared.

processhandle-2 input

INT .EXT:ref:10

is the other process handle to be compared.

Considerations

- PROCESSHANDLE_COMPARE_ considers two process handles to belong to the same process pair if they contain the same sequence number.

- PROCESSHANDLE_COMPARE_ compares only the contents of the input parameters; it does not send any messages.
- If either of the parameter supplied to PROCESSHANDLE_COMPARE_ is missing, the process terminates with instruction failure (trap 01).

PROCESSHANDLE_DECOMPOSE_ Procedure

Summary

The PROCESSHANDLE_DECOMPOSE_ procedure returns one or more parts of a process handle.

Syntax for C Programmers

Note. In the TNS/E environment the CEXTDECS file uses the *int* data type for 32-bit values. This is a change from the TNS and TNS/R environments where CEXTDECS uses the *long* data type for 32-bit values.

```
#include <cextdecs(PROCESSHANDLE_DECOMPOSE_)>

short PROCESSHANDLE_DECOMPOSE_ (
                              short *processhandle
                             ,[ short *cpu ]
                             ,[ short *pin ]
                             ,[ __int32_t *nodenumber ]
                             ,[ char *nodename ]
                             ,[ short maxlen ]
                             ,[ short *nodename-length ]
                             ,[ char *procname ]
                             ,[ short maxlen ]
                             ,[ short *procname-length ]
                             ,[ long long *sequence-number ] );
```

The character-string parameters *nodename* and *procname* are each followed by a parameter *maxlen* that specifies the maximum length in bytes of the character string and an additional parameter that returns the actual length of the string. In each case, the character-string parameter and the two parameters that follow it must either all be supplied or all be absent.

Syntax for TAL Programmers

```
error := PROCESSHANDLE_DECOMPOSE_ ( processhandle          !
i
                                  ,[ cpu ]                 !
o
                                  ,[ pin ]                 !
o
                                  ,[ nodenumber ]          !
o
                                  ,[ nodename:maxlen ]     !
o:i
                                  ,[ nodename-length ]     !
o
                                  ,[ procname:maxlen ]     !
o:i
                                  ,[ procname-length ]     !
o
                                  ,[ sequence-number ] ); !
o
```

Parameters

error returned value

INT

is a file-system error number indicating the outcome of the operation.

processhandle input

INT .EXT:ref:10

is the process handle from which one or more parts is returned.

cpu output

INT .EXT:ref:1

if present, returns the processor number of the process designated by *processhandle*.

pin output

INT .EXT:ref:1

if present, returns the process identification number of the process designated by *processhandle*.

nodenumber output

INT(32) .EXT:ref:1

if present, returns the number of the node in which the process designated by *processhandle* resides.

nodename:*maxlen* output:input

STRING .EXT:ref:*, INT:value

if present, returns the name of the node in which the process designated by *processhandle* resides.

maxlen is the length in bytes of the string buffer *nodename*.

nodename-length output

INT .EXT:ref:1

is the actual length of the value returned in *nodename*, in bytes.

procname:*maxlen* output:input

STRING .EXT:ref:*, INT:value

if present, returns the name of the process designated by *processhandle* if the process is named. The returned value is the simple name beginning with a dollar sign; it does not include a node name or ASCII sequence number.

maxlen is the length in bytes of the string buffer *procname*.

procname-length output

INT .EXT:ref:1

is the actual length of the value returned in *procname*, in bytes. For unnamed processes, *procname-length* is 0 and there is no error.

sequence-number output

FIXED .EXT:ref:1

if present, returns the sequence number from the specified process handle.

Considerations

If you specify *procname* or *procname-length*, and *processhandle* designates a named process, PROCESSHANDLE_DECOMPOSE_ looks up the process by name. If it does not exist, error 14 is returned.

Related Programming Manual

For programming information about the PROCESSHANDLE_DECOMPOSE_ procedure, see the *Guardian Programmer's Guide*.

PROCESSHANDLE_GETMINE_ Procedure

Summary

The PROCESSHANDLE_GETMINE_ procedure obtains the caller's process handle. For a caller that needs to obtain only its own process handle, a call to PROCESSHANDLE_GETMINE_ is more efficient than a call to PROCESS_GETINFO_.

For general information about process handles, see Appendix D, File Names and Process Identifiers.

Syntax for C Programmers

```
#include <cextdecs(PROCESSHANDLE_GETMINE_)>

short PROCESSHANDLE_GETMINE_ ( short *processhandle );
```

Syntax for TAL Programmers

```
error := PROCESSHANDLE_GETMINE_ ( processhandle );    ! o
```

Parameters

error returned value

INT

indicates the outcome of the operation. It returns one of these values:

0 Information returned successfully
3 Parameter address out of bounds

processhandle output

INT .EXT:ref:10

returns the caller's process handle.

Related Programming Manual

For programming information about the PROCESSHANDLE_GETMINE_ procedure, see the *Guardian Programmer's Guide*.

PROCESSHANDLE_NULLIT_ Procedure

Summary

The PROCESSHANDLE_NULLIT_ procedure initializes a process handle to a null value. A process handle that has -1 in each word is recognized by the operating system as being null.

For further information about process handles, see Appendix D, File Names and Process Identifiers.

Syntax for C Programmers

```
#include <cextdecs(PROCESSHANDLE_NULLIT_)>

short PROCESSHANDLE_NULLIT_ ( short *processhandle );
```

Syntax for TAL Programmers

```
error := PROCESSHANDLE_NULLIT_ ( processhandle );    ! o
```

Parameters

error returned value

INT

indicates the outcome of the operation. It returns one of these values:

0 Operation was successful.
22 Parameter is out of bounds.
29 Parameter is missing.

processhandle output

INT .EXT:ref:10

returns a null process handle (-1 in each word).

PROCESSHANDLE_TO_CRTPID_ Procedure

Summary

The PROCESSHANDLE_TO_CRTPID_ procedure converts a process handle to the corresponding process ID (CRTPID). For a description of process IDs, see Appendix D, File Names and Process Identifiers.

Syntax for C Programmers

```
#include <cextdecs(PROCESSHANDLE_TO_CRTPID_)>

short PROCESSHANDLE_TO_CRTPID_ ( short *processhandle
                                ,short *process-id
                                ,[ short pair-flag ]
                                ,[ __int32_t node-number ] );
```

- CEXTDECS (through the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
error := PROCESSHANDLE_TO_CRTPID_ ( processhandle       ! i
                                   ,process-id          ! o
                                   ,[ pair-flag ]       ! i
                                   ,[ node-number ] );  ! i
```

Parameters

error returned value

INT

is a file-system error number indicating the outcome of the operation.

processhandle input

INT .EXT:ref:10

is the process handle to be converted. An *error* value of 590 is returned if *processhandle* is null (-1 in each word) or has an invalid format.

process-id output

INT .EXT:ref:4

returns the process ID (CRTPID) of the process designated by *processhandle*. If the process is named and local to the node indicated by *node-number*, the process ID is in local form. In all other cases the process ID is in network form.

pair-flag input

INT:value

specifies whether *process-id* should designate a process pair (1 if it should; 0 if it should not). If *pair-flag* is set and the process is named, the *cpu* and *pin* values in *process-id* are set to -1 instead of the *cpu* and *pin* of the process. The default is 0.

node-number input

INT(32):value

if present and not -1D, identifies the node with respect to which *process-id* is normalized. If this parameter is omitted or -1D, the caller's node is used. See the *process-id* parameter.

Considerations

- If the name is longer than four characters (or five characters for local process) excluding the dollar sign, error 20 is returned.
- If the process is named, PROCESSHANDLE_TO_CRTPID_ looks up the process name in the destination control table (DCT). If the name is not found, error 14 is returned. However, it is sometimes possible for the name of a nonexistent process to be found in the DCT, in which case error 0 is returned. Therefore, even for a named process, error 0 (successful conversion of a process handle) does not guarantee that the process exists.
- If the PIN of the process is larger than 255, a synthetic process ID is returned along with an error 560. A synthetic process ID contains a PIN value of 255 in place of a high-PIN value, which cannot be represented by 8 bits.

PROCESSHANDLE_TO_FILENAME_ Procedure

Summary

The PROCESSHANDLE_TO_FILENAME_ procedure converts a process handle to a process file name.

Syntax for C Programmers

```
#include <cextdecs(PROCESSHANDLE_TO_FILENAME_)>

short PROCESSHANDLE_TO_FILENAME_ ( short *processhandle
                                  ,char *filename
                                  ,short maxlen
                                  ,short *filename-length
                                  ,[ short options ] );
```

Syntax for TAL Programmers

```
error := PROCESSHANDLE_TO_FILENAME_ ( processhandle      ! i
                                     ,filename:maxlen    !
o:i
                                     ,filename-length    ! o
                                     ,[ options ] );     ! i
```

Parameters

error returned value

INT

is a file-system error number indicating the outcome of the operation. If error 18 (unknown system) is returned, the process handle was converted except for the system name; "\255" is used for the system name.

processhandle input

INT .EXT:ref:10

is the process handle to be converted. If a null process handle (-1 in each word) is specified, the process handle of the calling process is used.

filename:*maxlen* output:input

STRING .EXT:ref:*, INT:value

returns the process file name of the process designated by *processhandle*. *filename* includes the node name of the process; it does not include qualifiers.

maxlen is the length in bytes of the string variable *filename*.

filename-length output

INT .EXT:ref:1

is the actual length of the value returned in *filename*. If an error other than 18 (unknown system) is returned, 0 is returned for this parameter.

options input

INT:value

specifies options. The fields are:

<0:14>	Not currently used (specify 0)
<15>	For named processes: if set, specifies that the sequence number not be included in *filename* for a named process. If this bit is not set, the sequence number is included. For unnamed processes: the sequence number is always included in *filename*, regardless of the value of this bit.

The default is 0.

Considerations

If the process is named, PROCESSHANDLE_TO_FILENAME_ looks up the process name in the destination control table (DCT). If the name is not found, error 14 is returned. However, it is sometimes possible for the name of a nonexistent process to be found in the DCT, in which case error 0 is returned. Therefore, even for a named process, error 0 (successful conversion of a process handle) does not guarantee that the process exists.

Related Programming Manual

For programming information about the PROCESSHANDLE_TO_FILENAME_ procedure, see the *Guardian Programmer's Guide*.

PROCESSHANDLE_TO_STRING_ Procedure

Summary

The PROCESSHANDLE_TO_STRING_ procedure converts a process handle to the equivalent process string. See “Considerations,” below, for a description of process strings.

Syntax for C Programmers

```
#include <cextdecs(PROCESSHANDLE_TO_STRING_)>

short PROCESSHANDLE_TO_STRING_( short *processhandle
                               ,char *process-string
                               ,short maxlen
                               ,short *process-string-length
                               ,[ char *nodename ]
                               ,[ short length ]
                               ,[ short named-form ] );
```

The parameter *length* specifies the length in bytes of the character string pointed to by *nodename*. The parameters *nodename* and *length* must either both be supplied or both be absent.

Syntax for TAL Programmers

```
error := PROCESSHANDLE_TO_STRING_ ( processhandle          ! i
                                   ,process-string:maxlen !
o:i
                                   ,process-string-length ! o
                                   ,[ nodename:length ]   !
i:i
                                   ,[ named-form ] );     ! i
```

Parameters

error returned value

INT

is a file-system error number indicating the outcome of the operation.

processhandle input

INT .EXT:ref:10

is the process handle to be converted.

process-string:*maxlen* output:input

STRING .EXT:ref:*, INT:value

returns a process string that represents the process designated by *processhandle*. The node name is included in *process-string*, except as described under *node*.

maxlen is the length in bytes of the string variable *process-string*.

process-string-length output

INT .EXT:ref

is the actual length of the value returned in *process-string*. If an error occurs, 0 is returned.

nodename:*length* input:input

STRING .EXT:ref:*, INT:value

if supplied and if *length* is not 0, specifies the node name that should be included in *process-string*. If used, the value of *nodename* must be exactly *length* bytes long.

If *nodename* designates the same node as indicated in *processhandle*, no node name is included in *process-string*. If it does not match the node indicated in *processhandle*, or if the parameter is omitted, or if *length* is 0, then the node name indicated in *processhandle* is included in *process-string*.

named-form input

INT:value

specifies the form of *process-string* to be returned for named processes. The *named-form* parameter is ignored for unnamed processes. Valid values are:

0 Return process name if possible; if it is unavailable, return *cpu*,*pin* form. See “Considerations.”
1 Return process name; if it is unavailable, report the error. See “Considerations.”
2 Return *cpu*,*pin* form in all cases.

A process name is unavailable if *processhandle* refers to a named process that no longer exists.

The default is 0.

Considerations

- A process string is a string of characters that identifies a process or a set of processes. Process strings are commonly used in command lines (for example, in the TACL STATUS command). PROCESSHANDLE_TO_STRING_ returns a process string in one of these forms:

  ```
  [\node.]cpu,pin
  [\node.]$process-name
  ```

- If you request the process name for a named process, PROCESSHANDLE_TO_STRING_ looks up the process by name. If the process does not exist and *named-form* is specified as 1, error 14 is returned.
- Conversion of the process handle does not necessarily include any check for the existence of the process; error 0 might be returned for a nonexistent process.

Related Programming Manual

For programming information about the PROCESSHANDLE_TO_STRING_ procedure, see the *Guardian Programmer's Guide*.

PROCESSINFO Procedure (Superseded by PROCESS_GETINFOLIST_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The PROCESSINFO procedure is used to obtain process status information.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
error := PROCESSINFO ( cpu,pin                      ! i
                      ,[ process-id ]               ! i,o
                      ,[ creator-access-id ]        ! i,o
                      ,[ process-access-id ]        ! i,o
                      ,[ priority ]                 ! i,o
                      ,[ program-filename ]         ! i,o
                      ,[ home-terminal ]            ! i,o
                      ,[ sysnum ]                   ! i
                      ,[ search-mode ]              ! i
                      ,[ priv-only ]                ! o
                      ,[ process-time ]             ! o
                      ,[ waitstate ]                ! o
                      ,[ process-state ]            ! o
                      ,[ library-filename ]         ! o
                      ,[ swap-filename ]            ! o
                      ,[ context-changes ]          ! o
                      ,[ flag ]                     ! o
                      ,[ licenses ]                 ! o
                      ,[ jobid ] );                 ! i,o
```

Parameters

error returned value

INT

returns a value indicating the outcome of the call.

0 Status for process *cpu,pin* is returned.

1 Process *cpu,pin* does not exist or does not match specified criteria (see *search-mode*). Status for next higher *cpu,pin* in the specified processor is returned. The 4-word process ID of the process for which status is being returned is returned, in the *process-id* parameter (if present).

2 Process *cpu,pin* does not exist, and no higher *cpu,pin* in the specified processor that matches the specified criteria exists (see *search-mode*).

3 Unable to communicate with *cpu*.

5 The system specified by *sysnum* could not be accessed.

6 Internal error.

7 Unable to process the D-series file name.

99 Parameter error.

cpu,pin input

INT:value

is the processor number (processor in bits <4:7> with <0:3> set to 0) and PIN (bits <8:15>) number of the process whose status is being requested. The process identification number (PIN) is a number used to uniquely identify the process control block (PCB) in a processor for a process.

process-id input, output

INT:ref:4

is an array where PROCESSINFO returns the 4-word process ID of the process whose status is actually being returned. This can be different from the process whose status is requested through *cpu,pin* (see the *error* parameter).

On input, the *process-id* contents can be used as a search criterion (see the *search-mode* parameter).

Note that process ID is a 4-word array where:

[0:2]	Process name or creation timestamp
[3].<0:3>	Reserved
.<4:7>	Processor number where the process is executing
.<8:15>	PIN assigned by the operating system to identify the process in the processor.

creator-access-id input, output

INT:ref:1

returns the creator access ID of *process-id*. The creator access ID identifies the user who initiates the creation of the process. For information about the creator access ID, see Considerations on page 3-151 and to the *Guardian User's Guide*.

On input, the *creator-access-id* contents can be used as a search criterion (see the *search-mode* parameter).

process-access-id input, output

INT:ref:1

returns the process access ID of *process-id*. For procedure PROCESSACCESSID, see the description under “Considerations” and to the *Guardian User’s Guide* for information about the process access ID.

On input, the *process-access-id* contents can be used as a search criterion (see the *search-mode* parameter).

priority input, output

INT:ref:1

returns the current execution priority of this process.

On input, the *priority* contents can be used as a search criterion (see the *search-mode* parameter).

program-filename input, output

INT:ref:12

is an array where PROCESSINFO returns the internal-format file name of the *process-id*’s program file.

On input, the contents of *program-filename* can be used as a search criterion (see the *search-mode* parameter). To designate a file that resides on a remote system designated by *sysnum*, you can simply specify the local form of the file name; if you specify the network form of the file name, the system number must match *sysnum* or an error is returned.

home-terminal input, output

INT:ref:12

is an array where PROCESSINFO returns the internal-format device name of the *process-id*’s home terminal.

On input, the *home-terminal* contents can be used as a search criterion (see the *search-mode* parameter).

sysnum input

INT:value

specifies the system (in a network) where the process for which information is to be returned is running. If this parameter is omitted, the local system is assumed.

search-mode input

INT:value

is a bit mask that specifies one or more “search” conditions.

The input values of certain parameters to PROCESSINFO are used as the search conditions; information is returned for the first process that matches the conditions.

The search is conducted on the processor specified in *cpu,pin*. The specified PIN is searched first and if it does not match the conditions, the higher PINs are progressively searched.

The bit fields in *search-mode* specify the conditions being searched for:

<0> = 1 must match *process-id* for 3 words
= 0 no search

<1> = 1 must match *creator-access-id*
= 0 no search

<2> = 1 must match *process-access-id*
= 0 no search

<3> = 1 must be <= *priority*
= 0 no search

<4> = 1 must match *program-filename*
= 0 no search

<5> = 1 must match *home-terminal*
= 0 no search

<6> = 1 must match *jobid*
= 0 no search

If multiple search conditions are specified, then all must be met.

If *search-mode* is omitted, the default value is 0.

priv-only output

INT:ref:*

This parameter can be used only by a privileged caller.

process-time output

FIXED:ref:1

returns the process time, in microseconds, for which the process has executed.

wait-state output

INT:ref:1

returns the wait field indicating what, if anything, the process is waiting on. It is obtained from the wait field of the awake/wait word in the process's process control block. These bits are defined:

<8> wait on PON (processor power on)
<9> wait on IOPON (I/O power on)
<10> wait on INTR (interrupt)
<11> wait on LINSP (Inspect event)
<12> wait on LCAN (message system, cancel)

<13> wait on LDONE (message system, done)
<14> wait on LTMF (TMF request)
<15> wait on LREQ (message system, request)

The bits in the wait field are numbered from left to right; thus, if octal 3 (%003) appears, this means that bits 14 and 15 are equal to 1.

process-state output

INT:ref:1

returns the state of the process specified by *cpu,pin*. The bits are defined as follows:

<0> privileged process
<1> page fault occurred
<2> process is on the ready list
<3> system process
<4:5> reserved
<6> memory access breakpoint in system code
<7> process not accepting any messages
<8> temporary system process
<9> process has logged on (called USER_AUTHENTICATE_ or VERIFYUSER)
<10> in a pending process state
<11:15> the process state, where:
0 unallocated process
1 starting
2 runnable
3 suspended
4 Debug memory access breakpoint
5 Debug breakpoint
6 Debug trap or signal
7 Debug request
8 Inspect memory access breakpoint
9 Inspect breakpoint
10 Inspect trap or signal
11 Inspect request
12 saveabend
13 terminating
14 XIO initialization

library-filename output

INT:ref:12

returns the internal-format file name of the library file used by the process. If the process does not have an associated library file, then *library-filename* is blank-filled.

swap-filename output

INT:ref:12

returns $*volume*.#0. Processes do not swap to $*volume*.#0; they swap to a swap file managed by the Kernel-Managed Swap Facility. For more information on this facility, see the *Kernel-Managed Swap Facility (KMSF) Manual*.

For TNS processes on RVUs preceding the D42 RVU, this parameter returns the internal-format file name of the swap file for the process's data segment. This is often the name of a temporary file unless a specific swap file is supplied at run time. It can also indicate the current swap volume.

context-changes output

INT:ref:1

gives the number of changes made to the DEFINE process context since process creation modulo 65,536. See "Considerations."

flag output

INT:ref:1

flag.<14:15> returns 0 if DEFINEs are disabled and returns 1 if DEFINEs are enabled.

licenses output

INT:ref:1

licenses.<15> returns 0 if the program file of the process was not licensed at process-creation time, and returns 1 if the program file of the process was licensed at process-creation time.

jobid input, output

INT:ref:5

consists of the GMOM's process ID plus the *jobid*. If this field is zero, the process does not belong to a job. If this field is nonzero, the GMOM's process ID identifies the ancestor of the job.

Considerations

- Remote or local form of *process-id*

  If *sysnum* specifies a remote system, *process-id* returns in network form; otherwise, *process-id* returns in local form. The two forms differ only in the form of the process name.

- A local process name consists of six bytes with the first byte being a dollar sign ($) and the second being an alphabetic character. The remaining four characters can be alphanumeric. Note that a full six character local process name cannot be converted to a remote form.

- A remote process name consists of six bytes with the first byte containing a backslash character (\). The second byte contains the number of the node where the process resides. The third must be an alphabetic character. The remaining three characters can be alphanumeric.
- Remote system *sysnum*

  If *sysnum* specifies a remote system, file names (such as home terminal) are passed in and returned in a form relative to the remote system. Local names (starting with $) are local to the remote system.
- Process DEFINE context changes

  Each process has an associated count of the changes to its context. This count is incremented each time the procedures DEFINEADD, DEFINEDELETE, and DEFINEDELETEALL are invoked and a consequent change to the process context occurs. In the case of DEFINEDELETE and DEFINEDELETEALL, the count is incremented by one even if more than one DEFINE is deleted. The count is also incremented if the DEFINE mode of the process is changed. If a call to CHECKDEFINE causes a DEFINE in the backup process to be altered, deleted, or added, then the count for the backup process is incremented. This count is 0 for newly created processes, and new processes do not inherit the count of their creators.
- High-PIN processes

  You cannot use PROCESSINFO on high-PIN processes, because a high PIN cannot fit into *cpu,pin* or *process-id*.

Example

```
CALL PROCESSINFO ( PID , PROCESSID , CAID , PAID , PRI , PROG
                   , HOMETERM , , MODE );
```

PROCESSNAME_CREATE_ Procedure

Summary

The PROCESSNAME_CREATE_ procedure returns a unique process name that is suitable for passing to the PROCESS_LAUNCH_, PROCESS_CREATE_, or PROCESS_SPAWN_ procedure. This type of naming (as opposed to using a predefined process name) is used when the name of a process pair does not need to be known to other processes in the system or network.

Syntax for C Programmers

```
#include <cextdecs(PROCESSNAME_CREATE_)>

short PROCESSNAME_CREATE_ ( char *name
                           ,short maxlen
                           ,short *namelen
                           ,[ short name-type ]
                           ,[ const char *nodename ]
                           ,[ short length ]
                           ,[ short options ] );
```

The parameter *length* specifies the length in bytes of the character string pointed to by *nodename*. The parameters *nodename* and *length* must either both be supplied or both be absent.

Syntax for TAL Programmers

```
error := PROCESSNAME_CREATE_ ( name:maxlen          ! o:i
                              ,namelen              ! o
                              ,[ name-type ]        ! i
                              ,[ nodename:length ]  ! i:i
                              ,[ options ] );       ! i
```

Parameters

error returned value

INT

indicates the outcome of the operation. It returns one of these file-system errors:

0	Process name is returned successfully.
44	No names of the specified type are available.
201	Unable to communicate with the specified node.
563	Output buffer is too small.
590	Parameter or bounds error.

name:maxlen output:input

STRING .EXT:ref:*, INT:value

returns the process name.

maxlen is the length in bytes of the string variable *name*.

namelen output

INT .EXT:ref:1

contains the actual length in bytes of the name being returned.

name-type input

INT:value

specifies the type of name desired. Values are:

0 4-character name
1 5-character name

If this parameter is omitted, 0 is used.

The local portion of the name is of the form $X*name*, $Y*name*, or $Z*name*, where *name* represents 1 to 4 alphanumeric character except “o” and “i.” This set of names is part of the set of process names that are reserved by the operating system. Applications should not use names of this form unless they have been obtained through this procedure.

The operating system reserved process name space includes these names: $X*name*, $Y*name*, and $Z*name*, where *name* is 1 to 4 alphanumeric characters. This set of names is also part of the set of process names that are reserved by the operating system. Applications should not use names of this form unless they have been obtained through this procedure.

nodename:length input:input

STRING .EXT:ref:*, INT:value

if supplied and if *length* is not 0, specifies the node name that is to be returned as part of the process name if a node name is desired, as indicated by the *options* parameter. If used, the value of *nodename* must be exactly *length* bytes long. If this parameter is omitted or if *length* is 0, and if *options*.<15> = 0 (node name is desired), the name of the caller’s node is used. See the *options* parameter.

options input

INT:value

can have these values:

<0:14>		Reserved; must be 0.
<15>	0	Include node name in the returned process name.

1 Return the process name in local form.

If this parameter is omitted, 0 is used.

Example

```
INT type := 1;  ! return a 5-character name
INT form := 1;  ! return name in local form
       .
       .
       .
err := PROCESSNAME_CREATE_ ( name:max^length, actual^length,
                             type, , form );
IF err THEN ...
```

Related Programming Manual

For programming information about the PROCESSNAME_CREATE_ procedure, see the *Guardian Programmer's Guide*.

PROCESSOR_GETINFOLIST_ Procedure

Summary

The PROCESSOR_GETINFOLIST_ procedure obtains configuration information and statistics about a processor. The processor of interest is specified by node name and processor number.

For further information about supported processors, see Table 12-6, Summary of Processor Types and Models.

Syntax for C Programmers

```
#include <cextdecs(PROCESSOR_GETINFOLIST_)>

short PROCESSOR_GETINFOLIST_ ( [ const char *nodename ]
                              ,[ short length ]
                              ,[ short cpu ]
                              ,short *ret-attr-list
                              ,short ret-attr-count
                              ,short *ret-values-list
                              ,short ret-values-maxlen
                              ,short *ret-values-len
                              ,[ short *error-detail ] );
```

The parameter *length* specifies the length in bytes of the character string pointed to by *nodename*. The parameters *nodename* and *length* must either both be supplied or both be absent.

Syntax for TAL Programmers

```
error := PROCESSOR_GETINFOLIST_ ( [ nodename:length ]  ! i:i
                                 ,[ cpu ]              ! i
                                 ,ret-attr-list        ! i
                                 ,ret-attr-count       ! i
                                 ,ret-values-list      ! o
                                 ,ret-values-maxlen    ! i
                                 ,ret-values-len       ! o
                                 ,[ error-detail ] );  ! o
```

Parameters

error returned value

INT

indicates the outcome of the operation. It returns one of these values:

error	Description
0	Information is returned for the specified process.
1	File-system error; *error-detail* contains the error number. Error 563 is returned if the *ret-values-list* buffer is too small to contain all of the requested information.
2	Parameter error; *error-detail* contains the number of the first parameter found to be in error, where 1 designates the first parameter on the left.
3	Bounds error; *error-detail* contains the number of the first parameter found to be in error, where 1 designates the first parameter on the left.
4	(Reserved)
5	Unable to communicate with *cpu*. *cpu* might not exist.
6	Unable to communicate with *nodename*.
7	An invalid return attribute code was supplied.

nodename:*length* input:input

STRING .EXT:ref:*, INT:value

if present and if *length* is not 0, specifies the name of the node that contains the processor of interest. *nodename* must be exactly *length* bytes long. If *nodename*:*length* is omitted or if *length* is 0, the name of the local node is used.

cpu input

INT:value

if present and if not -1, is the number of the processor of interest. If *cpu* is omitted, then the caller's processor number is used. In that case, *nodename*:*length* must be omitted.

ret-attr-list input

INT .EXT:ref:*

is an array of INTs indicating the attributes that are to have their values returned in *ret-values-list*.

ret-attr-count input

INT:value

indicates how many items the caller is supplying in *ret-attr-list*.

If the return values cannot fit into *ret-values-list*, the procedure returns an *error* of 1 and an *error-detail* value of 563 (buffer too small). No processor information is returned.

ret-values-list output

INT .EXT:ref:*

contains *ret-values-len* words of returned information. The values parallel the items in *ret-attr-list*. For details, see Attribute Codes and Value Representations on page 12-75. Each value begins on a word boundary. A value that is returned in the form of an array begins with an INT giving the number of elements in the array, followed by the actual array.

ret-values-maxlen input

INT:value

is the maximum length, in words, of *ret-values-list*.

ret-values-len output

INT .EXT:ref:1

is the actual length, in words, of *ret-values-list*.

error-detail output

INT .EXT:ref:1

for some returned errors, contains additional information. See *error*.

Note. Calls to this procedure are identical in their format and values to calls to CPU_GETINFOLIST_.

General Considerations

- If PROCESSOR_GETINFOLIST_ returns a nonzero *error* value, the contents of *ret-values-list* and *ret-values-len* are undefined.

Attribute Codes and Value Representations

The individual attribute codes and their associated TAL value representations are as follows:

Code	Attribute	TAL Value Representation
2	processor type	INT
3	software version	INT
4	page size	INT(32)
5	memory size	INT(32)
6	first virtual page	INT(32)
7	swappable pages	INT(32)
8	free pages	INT(32)
9	current locked memory	INT(32)
10	maximum locked memory	INT(32)
11	high locked memory	INT(32)
12	page faults	unsigned INT(32)
13	scans per memory manager call	INT(32)
14	memory clock cycles	unsigned INT(32)
15	memory pressure	INT
16	memory queue length	INT
17	system coldload time	FIXED
18	elapsed time	FIXED
19	busy time	FIXED
20	idle time	FIXED
21	interrupt time	FIXED
22	processor queue length	INT
23	dispatches	unsigned INT(32)
24	PCBs in low PINs	INT number of elements, INT ARRAY
25	PCBs in high PINs	INT number of elements, INT ARRAY
26	time list elements	INT number of elements, INT(32) ARRAY

Code	Attribute	TAL Value Representation
27	process time list elements	INT number of elements, INT(32) ARRAY
28	breakpoints	INT
29	send busy	FIXED
35	interrupt count	INT number of elements, INT(32) ARRAY
36	disk cache hits	FIXED
37	disk I/Os	FIXED
38	processor queue state	INT, INT, FIXED
39	memory queue state	INT, INT, FIXED
40	sequenced sends	unsigned INT(32)
41	unsequenced sends	unsigned INT(32)
42	CME events	unsigned INT(32)
43	pages created	unsigned INT(32)
44	interpreter busy	FIXED
45	interpreter transitions	INT(32)
46	transactions	unsigned INT(32)
47	processor model	INT
48	processor name	INT bytelength, STRING
49	processor full name	INT bytelength, STRING
50	accelerated time	FIXED
51	clock resolution	FIXED
52	maximum clock adjustment	FIXED
53	maximum clock drift	FIXED
54	clock sets	INT
55	system loads	INT
56	base time	FIXED
57	memory-management attributes	INT(32)
58	segments in use	INT(32)
59	maximum segments used	INT(32)
60	updates part of the release ID (the two digits that follow the period)	INT
61	internal use only	
62	availability of IEEE floating point on the current system	INT

Code	Attribute	TAL Value Representation
65	64-bit dispatch count	unsigned INT(64)
72	system name	INT bytelength, STRING
74	number of IPUs in a CPU	INT
78	Is this a NEO CPU?	INT
79	Current configured TLE limit	INT(32)

If PROCESSOR_GETINFOLIST_ cannot obtain meaningful data for an attribute that returns an array, it returns a value of 0 for the number of array elements and allocates no space for the actual array. Except where otherwise noted, PROCESSOR_GETINFOLIST_ returns a value of -1 (for an INT), -1D (for an INT(32)), or -1F (for a FIXED) when it cannot obtain a meaningful value for an attribute that does not return an array.

- 2: processor type

  See Table 12-6 on page 12-237 for processor type values.

- 3: software version

  the version of the operating system that is running. This value has the same format as the value returned by the TOSVERSION procedure. See TOSVERSION Procedure.

- 4: page size

  the page size of physical memory, in bytes.

- 5: memory size

  the size of physical memory, in pages. If the number of pages exceeds 2,147,483,647 (2**31 - 1), the returned value is -1D.

- 6: first virtual page

  the page number of the first swappable page.

- 7: swappable pages

  the current number of memory pages that can be swapped. If the number of pages exceeds 2,147,483,647 (2**31 - 1), the returned value is -1D.

- 8: free pages

  the current number of nonallocated memory pages. If the number of pages exceeds 2,147,483,647 (2**31 - 1), the returned value is -1D.

- 9: current locked memory

  the current amount of virtual memory, in pages, that is locked in physical memory. If the number of pages exceeds 2,147,483,647 (2**31 - 1), the returned value is -1D.

- 10: maximum locked memory

  the maximum amount of virtual memory, in pages, that can be locked in physical memory. If the number of pages exceeds 2,147,483,647 (2**31 - 1), the returned value is -1D.

- 11: high locked memory

  the maximum amount of virtual memory, in pages, that has been locked in physical memory at any one time since the processor was loaded. If the number of pages exceeds 2,147,483,647 (2**31 - 1), the returned value is -1D.

- 12: page faults

  the number of page-fault interrupts since the processor was loaded. This number is returned as an unsigned value.

- 13: scans per memory manager call

  during a call to the memory manager, the average number of pages, multiplied by 100, examined before one is found that can be deallocated.

- 14: memory clock cycles

  the number of times the memory manager has looked at all swappable pages of memory since the processor was loaded. This number is returned as an unsigned value.

- 15: memory pressure

  an indicator of the frequency of page faults. This number is in the range of 0 (low frequency) through 7 (high frequency).

- 16: memory queue length

  the current number of processes waiting for a page fault to be serviced. The returned value is an unsigned integer.

- 17: system coldload time

  the time (Greenwich mean time, or Coordinated Universal Time) at which the system was cold loaded.

- 18: elapsed time

  the amount of time, in microseconds, since the processor was loaded.

- 19: busy time

  the amount of time, in microseconds, that processes have been executing since the processor was loaded.

- 20: idle time

  the amount of time, in microseconds, that has not been spent in process execution or interrupt handling since the processor was loaded.

- 21: interrupt time

  the amount of time, in microseconds, that has been spent handling interrupts since the processor was loaded.

- 22: processor queue length

  the current number of processes that are ready to execute. The returned value is an unsigned integer.

- 23: dispatches

  the number of dispatch interrupts since the processor was loaded. This number is returned as an unsigned value.

- 24: PCBs in low PINs

  an array of counters that refer to the number of low-PIN process control blocks (PCBs) in the processor. The number of array elements is always 4 and the elements in the array are: maximum number used, current number in use, number free, and number of allocation failures.

- 25: PCBs in high PINs

  an array of counters that refer to the number of high-PIN process control blocks (PCBs) in the processor. The number of array elements is always 4 and the elements in the array are: maximum number used, current number in use, number free, and number of allocation failures.

- 26: time list elements

  an array of counters that refer to the number of time list elements (TLEs) for the processor. The number of array elements is always 4 and the elements in the array are: maximum number used, current number in use, number configured, and number of allocation failures.

- 27: process time list elements

  an array of counters that refer to the number of process time list elements (PTLEs) for the processor. The number of array elements is always 4 and the elements in the array are: maximum number used, current number in use, number configured, and number of allocation failures.

- 28: breakpoints

  the number of processor breakpoints currently set.

- 29: send busy

  the amount of time, in microseconds, that has been spent performing message sends since the processor was loaded.

- 35: interrupt count

an array of counters for the various interrupts. The number of array elements is 24 and the elements in the array are:

```
[0]   G-series:  0
[1]   uncorrectable memory error
[2]   memory access breakpoint
[3]   instruction failure
[4]   page fault
[5]   (reserved)
[6]   (reserved)
[7]   G-series:  0
[8]   power failure
[9]   correctable memory error
[10]  G-series:  0
[11]  G-series:  IPC traffic interrupts
[12]  (reserved)
[13]  time list
[14]  G-series:  IO traffic interrupts
[15]  dispatcher
[16]  power on
[17]  memory stack overflow
[18]  arithmetic overflow
[19]  instruction breakpoint
[20]  (reserved)
[21]  (reserved)
[22]  (reserved)
[23]  (reserved)
```

If the interrupt structure of the processor of interest cannot be mapped onto the T16 interrupt list, a value of 0 is returned for the number of array elements and no space is allocated for the actual array.

- 36: disk cache hits

  the number of times the disk processes have found desired disk blocks in memory since the processor was loaded.

- 37: disk I/Os

  the number of physical I/Os issued to the processor's disks since the processor was loaded.

- 38: processor queue state

  an array of two integers and a FIXED. The first integer is the maximum number of processes ready to run at any time since the Measure product started collecting statistics on the processor. The second integer is the current number of processes ready to run. The FIXED contains the total number of microseconds that all processes have spent on the ready queue.

- 39: memory queue state

an array of two integers and a FIXED. The first integer is the maximum number of processes waiting for memory at any time since the Measure product started collecting statistics on the processor. The second integer is the current number of processes waiting for memory. The FIXED contains the total number of microseconds that all processes have spent waiting

- 40: sequenced sends

  the number of message packets containing interprocess messages that have been sent since the processor was loaded. This number is returned as an unsigned value.

- 41: unsequenced sends

  the number of message packets containing low level control information that have been sent since the processor was loaded. This number is returned as an unsigned value.

- 42: CME events

  the number of correctable memory errors that have been detected since the processor was loaded. This number is returned as an unsigned value.

- 43: pages created

  the number of pages of virtual memory created by the processor's memory manager since the processor was loaded. This number is returned as an unsigned value.

- 44: interpreter busy

  for NSR-L processors, the amount of processor time in microseconds that the processor has spent in the interpreter since the Measure product started collecting statistics on the processor. If the processor is not a NSR-L processor, 0F is returned.

- 45: interpreter transitions

  for NSR-L processors, the number of times that accelerated code has entered the interpreter since the Measure product started collecting statistics on the processor. If the processor is not a NSR-L processor, 0D is returned.

- 46: transactions

  the number of transactions since the Measure product started collecting statistics on the processor. If the Measure product is not collecting statistics on the processor, 0D is returned. A transaction is defined as a read from a terminal followed by a write to a terminal. This number is returned as an unsigned value.

- 47: processor model

  the processor model number. Processor model numbers are defined only for certain processors. The processor model number is set to 0 when it is unknown,

undefined, or if the processor of interest is running on a RVU earlier than D20. See Table 12-6 on page 12-237 for processor model values.

- 48: processor name

  the processor name. See Table 12-6 on page 12-237 for processor name STRING values.

- 49: processor full name

  the processor full name. See Table 12-6 on page 12-237 for processor full name STRING values.

- 50: accelerated time

  the number of microseconds the processor spent in accelerated code since the Measure product started collecting statistics on the processor. If the Measure product is not collecting statistics on the processor or if the processor of interest is running a RVU earlier than D20, -1F is returned. If the processor of interest is not a native processor, then 0F is returned.

- 51: clock resolution

  the resolution of the system clock in nanoseconds.

- 52: maximum clock adjustment

  the maximum rate, in nanoseconds per second, that the system clock can be adjusted. This rate can be exceeded when the system clock is moved forward.

- 53: maximum clock drift

  the maximum rate, in nanoseconds per second, that the system clock can drift.

- 54: clock sets

  the number of times the clock was set since the processor was loaded.

- 55: system loads

  the number of system loads from the $SYSTEM disk.

- 56: base time

  the timestamp of when the processor was loaded. For a description of this form of the timestamp, see TIMESTAMP Procedure. The base time is set to -1F when the processor of interest is running a RVU earlier than D30.

- 57: memory-management attributes

  the memory-management attributes of the processor. These attributes are returned as a bit mask defined as:

<0:30>		Reserved
<31>	1	Flat segments supported
	0	Flat segments not supported

Flat segments are supported on native processors that use D30 or later RVUs of the NonStop operating system.

- 58: segments in use

  the number of absolute unitary segments currently in use. A unitary segment is a virtual memory area consisting of 128 kilobytes. It is the unit of virtual space allocation used by the NonStop operating system.

- 59: maximum segments used

  the maximum segments used since the last system load.

- 60: update part of the release ID (the two digits that follow the period)

  a binary number representing the two digits that follow the period (.) in the release identifier.

- 61: for HP internal use only

- 62: availability of IEEE floating point on the current system

  this attribute can be identified as CPUINFO_ATTR_FP_IEEE_VER, and can have these values:

0	No IEEE floating point
1	First version of IEEE floating-point support.
>1	Reserved for future versions of IEEE floating-point support.

- 65: 64-bit dispatch count

  the number of dispatch interrupts since the processor was loaded in a 64-bit counter.

- 72: system name

  the system name. See Table 12-6 on page 12-236 for system name STRING values.

- 74: number of IPUs in a CPU

  the number of IPUs in the specified CPU.

- 78: Is this a NEO CPU?

  returns 1 if the CPU is a part of the NeoView “segment”, or else the value returned is 0.

- 79: Current configured TLE limit

returns the current configured TLE limit.

Note. Attribute code 79 is available only for systems running J06.03 and later J-series RVUs or H06.14 and later H-series RVUs.

Example

In this example, the processor type and model of the caller's processor are returned in a structure.

```
LITERAL type = 2;
LITERAL model = 47;
INT attributes [0:1] := [ type, model ];
STRUCT processor^info;
   BEGIN
   INT processor^type;
   INT processor^model;
   END;
    .
    .
    .
error := PROCESSOR_GETINFOLIST_ ( nodename:length
                                 ,! cpu parameter not
needed,!
                                  ! defaults to caller's cpu
!
                                 ,attributes
                                 ,$OCCURS( attributes )
                                 ,processor^info
                                 ,$LEN ( processor^info ) / 2
                                 ,return^length );
```

Table 12-6. Summary of Processor Types and Models (page 1 of 4)

Processor type (Code 2)	Processor Model Value (Code 47)	Processor Model Name	Processor Name (Code 48)	Processor Full Name (Code 49)	System Name (Code 72)
0	0	0	NonStop 1+	HP NonStop 1+ CPU*	NonStop 1+
1	0	0	NonStop II	HP NonStop II CPU*	NonStop II
2	0	0	TXP	HP NonStop TXP CPU*	NonStop TXP
3	0	0	VLX	HP NonStop VLX CPU*	NonStop VLX
4	0	CLX	CLX	HP NonStop CLX CPU*	NonStop CLX
4	1	CLX 600	CLX	HP NonStop CLX 600 CPU*	NonStop CLX
4	2	CLX 700	CLX	HP NonStop CLX 700 CPU*	NonStop CLX
4	3	CLX 800	CLX	HP NonStop CLX 800 CPU*	NonStop CLX
4	3	CLX 800	CLX	HP NonStop CLX 800 CPU*	NonStop CO-CLX800
4	0 or 3	0 or CLX 800	NSR-L or CLX	HP NonStop System RISC Model L CPU or HP NonStop CLX 800 CPU	NonStop K100
5	0	0	Cyclone	HP NonStop Cyclone CPU*	NonStop Cyclone
6	0	0	NSR-L	HP NonStop System RISC Model L CPU*	NonStop CLX/R
6	0	0	NSR-L	HP NonStop System RISC Model L CPU*	NonStop CLX 2000
6	0	0	NSR-L	HP NonStop System RISC Model L CPU*	NonStop CO-Cyclone/R
6	0	0	NSR-L	HP NonStop System RISC Model L CPU*	NonStop Cyclone/R

* This system is no longer supported.

**Supported only on systems running H06.17 and later H-series RVUs and J06.06 and later J-series RVUs

Table 12-6. Summary of Processor Types and Models (page 2 of 4)

Processor type (Code 2)	Processor Model Value (Code 47)	Processor Model Name	Processor Name (Code 48)	Processor Full Name (Code 49)	System Name (Code 72)
6	0	0	NSR-L	HP NonStop System RISC Model L CPU*	NonStop K120
6	0	0	NSR-L	HP NonStop System RISC Model L CPU*	NonStop K1000
6	0	0	NSR-L	HP NonStop System RISC Model L CPU*	NonStop K1000SE
7	2	NSR-N	NSR-N	HP NonStop System RISC Model N CPU*	NonStop K10000
7	3	NSR-P	NSR-P	HP NonStop System RISC Model P CPU*	NonStop K20000
7	4	NSR-K	NSR-K	HP NonStop System RISC Model K CPU*	NonStop K200
7	4	NSR-K	NSR-K	HP NonStop System RISC Model K CPU*	NonStop K2000
7	4	NSR-K	NSR-K	HP NonStop System RISC Model K CPU*	NonStop K2000SE
8	0	NSR-W	NSR-W	HP NonStop System RISC Model W CPU	NonStop S7000
9	0	NSR-G	NSR-G	HP NonStop System RISC Model G CPU	NonStop S70000
9	1	NSR-T	NSR-T	HP NonStop System RISC Model T CPU	NonStop S72000
9	2	NSR-V	NSR-V	HP NonStop System RISC Model V CPU	NonStop S74000

* This system is no longer supported.

**Supported only on systems running H06.17 and later H-series RVUs and J06.06 and later J-series RVUs

Table 12-6. Summary of Processor Types and Models (page 3 of 4)

Processor type (Code 2)	Processor Model Value (Code 47)	Processor Model Name	Processor Name (Code 48)	Processor Full Name (Code 49)	System Name (Code 72)
9	3	NSR-X	NSR-X	HP NonStop System RISC Model X CPU	NonStop S76000
9	4	NSR-Y	NSR-Y	HP NonStop System RISC Model Y CPU	NonStop S86000
9	5	NSR-Z	NSR-Z	HP NonStop System RISC Model Z CPU	NonStop S88000
9	6	NSR-R	NSR-R	HP NonStop System RISC Model R CPU	NonStop S86100
9	7	NSR-S	NSR-S	HP NonStop System RISC Model S CPU	NonStop S7800B
9	10(A)	NSR-D	NSR-D	HP NonStop System RISC Model D CPU	NonStop S7400
9	11(B)	NSR-E	NSR-E	HP NonStop System RISC Model E CPU	NonStop S7600
9	13(D)	NSR-H	NSR-H	HP NonStop System RISC Model H CPU	NonStop S78000
9	14(E)	NSR-J	NSR-J	HP NonStop System RISC Model J CPU	NonStop S7800
10	1	NSE-A	NSE-A	HP NonStop System EPIC Model A CPU	HP Integrity NonStop NS16000
10	2	NSE-D	NSE-D	HP NonStop System EPIC Model D CPU	HP Integrity NonStop NS14000
10	11	NSE-B	NSE-B	HP NonStop System EPIC Model B CPU	HP Integrity NonStop NS1000

* This system is no longer supported.

**Supported only on systems running H06.17 and later H-series RVUs and J06.06 and later J-series RVUs

Table 12-6. Summary of Processor Types and Models (page 4 of 4)

Processor type (Code 2)	Processor Model Value (Code 47)	Processor Model Name	Processor Name (Code 48)	Processor Full Name (Code 49)	System Name (Code 72)
10	12	NSE-C	NSE-C	HP NonStop System EPIC Model C CPU	HP Integrity NonStop NEOVIEW
10	51	NSE-I	NSE-I	HP NonStop System EPIC Model I CPU	HP Integrity NonStop NS5000
10	63	NSE-K	NSE-K	HP NonStop System EPIC Model K CPU	HP Integrity NonStop NS3000AC
10	64	NSE-O	NSE-O	HP NonStop System EPIC Model O CPU	HP Integrity NonStop NEOVIEW
10	66	NSE-X	NSE-X	HP NonStop System EPIC Model X CPU	HP Integrity NonStop NEOVIEW
10	67	NSE-W	NSE-W	HP NonStop System EPIC Model W CPU **	HP Integrity NonStop NS2000 **
10	71	NSE-M	NSE-M	HP NonStop System EPIC Model M CPU	HP Integrity NonStop NB50000c
10	82	NSE-S	NSE-S	HP NonStop System EPIC Model S CPU	HP Integrity NonStop NS14200
10	83	NSE-T	NSE-T	HP NonStop System EPIC Model T CPU	HP Integrity NonStop 16200
10	91	NSE-Q	NSE-Q	HP NonStop System EPIC Model Q CPU	HP Integrity NonStop NS1200
10	92	NSE-R	NSE-R	HP NonStop System EPIC Model R CPU	HP Integrity NonStop NS3200AC

* This system is no longer supported.

**Supported only on systems running H06.17 and later H-series RVUs and J06.06 and later J-series RVUs

PROCESSOR_GETNAME_ Procedure

Summary

The PROCESSOR_GETNAME_ procedure returns a processor's type and model. You can designate the processor of interest either by supplying a processor number with a node number or name, or by supplying a processor number alone. Alternatively, you can supply just the numeric representation of the processor type. If none of these are supplied, the procedure returns information about the caller's processor.

For further information about supported processors, see Table 12-6, Summary of Processor Types and Models.

Syntax for C Programmers

```
#include <cextdecs(PROCESSOR_GETNAME_)>

short PROCESSOR_GETNAME_ ( short cpu-number
                          ,char *name
                          ,short maxlen
                          ,short *namelen
                          ,[ short *cpu-type-out ]
                          ,[ const char *node-name ]
                          ,[ short length ]
                          ,[ __int32_t node-number ]
                          ,[ short cpu-type-in ]
                          ,[ short expand-name ]
                          ,[ short *cpu-model-out ]
                          ,[ short cpu-model-in ] );
```

- CEXTDECS (through the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.
- The parameter *length* specifies the length in bytes of the character string pointed to by *node-name*. The parameters *node-name* and *length* must either both be supplied or both be absent.

Syntax for TAL Programmers

```
error := PROCESSOR_GETNAME_ ( [ cpu-number ]          ! i
                             ,name:maxlen             ! o:i
                             ,namelen                 ! o
                             ,[ cpu-type-out ]        ! o
                             ,[ node-name:length ]    ! i:i
                             ,[ node-number ]         ! i
                             ,[ cpu-type-in ]         ! i
                             ,[ expand-name ]         ! i
                             ,[ cpu-model-out ]       ! o
                             ,[ cpu-model-in ] );     ! i
```

Parameters

error returned value

INT

returns a file-system error number indicating the outcome of the operation. Possible values include:

0	Information returned successfully
22	Parameter or buffer out of bounds
29	Missing parameter
201	Unable to communicate over this path
590	Parameter value bad or inconsistent

cpu-number input

INT:value

is the number that identifies the processor of interest. This parameter is required when either *node-name* or *node-number* is specified. If *cpu-number* is omitted or equal to -1, and if neither *node-name* nor *node-number* is specified, then the caller's processor is used.

name:maxlen output:input

STRING .EXT:ref:*, INT:value

returns the processor type as a character string. *maxlen* specifies the length in bytes of the string variable *name* and must be at least 3. If the name to be returned is longer than *maxlen*, the returned value is truncated to *maxlen* bytes. If the processor type is unknown, the procedure returns a blank string in *name* and 0 in *namelen*.

Possible return values for *name* are:

Processor Type	***name***
0	“NonStop 1+”
1	“NonStop II”
2	“TXP”
3	“VLX”
4	“NSR-L” or “CLX”
5	“Cyclone”
6	“NSR-L”
7	“NSR-N” “NSR-P” “NSR-K”
8	“NSR-W”
9	“NSR-D”
9	“NSR-E”
9	“NSR-G”
9	“NSR-H”
9	“NSR-J”
9	"NSR-T"
9	“NSR-V”
9	“NSR-X”
9	“NSR-Y”
9	“NSR-Z”
10	“NSE-A”
otherwise	*maxlen* blanks

Processor types 0, 1, 2, 3, 4, and 5 are no longer supported.

namelen output

INT .EXT:ref:1

returns the actual length in bytes of the value returned in *name*. 0 is returned if an error occurs.

cpu-type-out output

INT .EXT:ref:1

returns the processor type in numeric form. The possible values are shown earlier under the description of the *name* parameter (see the column labeled “Processor

Type”). These are the same values that are returned by the PROCESSORTYPE procedure.

node-name:*length* input:input

STRING .EXT:ref:*, INT:value

if present and if *length* is not equal to 0, specifies the name of the node where the processor of interest is located. The value of *node-name* must be exactly *length* bytes long. If this parameter is omitted or if *length* is 0, and if *node-number* does not specify a node, the local node is used.

node-number input

INT(32):value

if present and if not equal to -1D, specifies the number of the node where the processor of interest is located. If this parameter is omitted or equal to -1D, and if *node-name* does not specify the node, the local node is used.

cpu-type-in input

INT:value

if present and if not equal to -1, specifies the processor type in numeric form. This value must be one of the numeric values shown earlier under the description of the *name* parameter (see the column labeled “Processor Type”).

expand-name input

INT:value

if present and equal to 1, causes the returned value in *name* to be expanded. For most processor types, the returned value becomes “HP NonStop *name* CPU.” For the NonStop 1+ and the NonStop II processors, the word “NonStop” is not repeated. For the NSR-L processor, the name is expanded to “HP NonStop RISC Model L.”

cpu-model-out output

INT .EXT:ref:1

returns the processor model number of the processor returned in the *cpu-type-out* parameter. For a list of model numbers, see Table 12-6 on page 12-237.

cpu-model-in input

INT:value

specifies the processor model number of the processor specified in the *cpu-type-in* parameter. For a list of model numbers, see Table 12-6 on page 12-237.

Considerations

If you supply more information than is necessary to identify the processor or the processor type of interest (that is, if you specify both *node-number* and *node-name*, or if you identify the processor and also specify *cpu-type-in*), PROCESSOR_GETNAME_ uses the first sufficient set of parameters that it encounters and ignores the rest.

Example

In this example, the processor of interest is identified by its processor number and node number.

```
error := PROCESSOR_GETNAME_ ( cpu^num, name:max^length,
                              length, , , node^num );
```

PROCESSORSTATUS Procedure

Summary

The PROCESSORSTATUS procedure returns the highest processor number plus 1 of the configured processor modules in a system and the operational states of all the processor modules.

For further information about supported processors, see Table 12-6, Summary of Processor Types and Models, on page 12-237.

Syntax for C Programmers

```
#include <cextdecs(PROCESSORSTATUS)>

__int32_t PROCESSORSTATUS ();
```

- CEXTDECS (through the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
processor-status := PROCESSORSTATUS;
```

Parameters

processor-status returned value

INT(32)

returns two words indicating the highest processor number plus 1 of the configured processor modules and the operational states of all the processor modules.

The most significant word contains the highest processor number plus one.

The least significant word is a bit mask indicating the operational state of each processor module:

Word[0]	most significant word, highest processor number + 1
[1]	least significant word, bit mask 1 or 0

ls word.<0> = processor module 0
ls word.<1> = processor module 1
.
.
.
ls word.<14> = processor module 14
ls word.<15> = processor module 15

For each bit:

1	up	indicates that the corresponding processor module is up (operational).
0	down	indicates that the corresponding processor module is down or does not exist.

PROCESSORTYPE Procedure

Summary

The PROCESSORTYPE procedure returns the processor type of a specified system and processor.

For further information about supported processors, see Table 12-6, Summary of Processor Types and Models, on page 12-237.

Syntax for C Programmers

```
#include <cextdecs(PROCESSORTYPE)>

short PROCESSORTYPE ();
```

Syntax for TAL Programmers

```
type := PROCESSORTYPE ( [ cpu ]                    ! i
                      ,[ sysid ] );                ! i
```

Parameters

type returned value

INT

returns one of these values:

-2 feature not supported for the system named in *sysid*
-1 unable to communicate with processor (either it does not exist or the network is down)
0 HP NonStop 1+ processor
1 HP NonStop II processor
2 HP NonStop TXP processor
3 HP NonStop VLX processor
4 HP NonStop CLX processor
5 HP NonStop Cyclone processor
6 HP NonStop NSR-L processor
7 HP NonStop NSR-N processor,
HP NonStop NSR-P processor, or
HP NonStop NSR-K processor
8 HP NonStop NSR-W processor
9 HP NonStop NSR-D processor
HP NonStop NSR-E processor
HP NonStop NSR-G processor
HP NonStop NSR-H processor
HP NonStop NSR-J processor
HP NonStop NSR-T processor
HP NonStop NSR-V processor
HP NonStop NSR-X processor
HP NonStop NSR-Y processor
HP NonStop NSR-Z processor
10 HP NonStop NSE-A processor

If *cpu* is greater than 16 or less than 0, then -1 is returned. If *sysid* is invalid or the system is unavailable across the network, then -1 is returned. Types 0, 1, 2, and 3 are no longer supported.

cpu input

INT:value

is the processor number of the processor for the *type* returned.

If no value is specified for *cpu*, the processor from which the call is made is used and the *sysid* parameter is ignored.

sysid input

INT:value

is the system number, identifying the system of the processor of which the type is returned. If no value is specified for *sysid* the system from which the call is made is used.

Example

```
TYPE^CPU := PROCESSORTYPE ( PROCESSOR , SYSTEM^NUM );
```

PROCESSSTRING_SCAN_ Procedure

Summary

The PROCESSSTRING_SCAN_ procedure scans an input string for a process string and returns the corresponding process handle or a single component of the process string converted to internal form. Device names are optionally accepted in the input string. See “Considerations” for the definition of process string.

Syntax for C Programmers

```
#include <cextdecs(PROCESSSTRING_SCAN_)>

short PROCESSSTRING_SCAN_ ( char *string
                           ,[ short length_of_searchString ]
                           ,[ short *length-used ]
                           ,[ short *processhandle ]
                           ,[ short *stringtype ]
                           ,[ char *name ]
                           ,[ maxlen ]
                           ,[ short *namelen ]
                           ,[ short *cpu ]
                             ,[ short *pin ]
                             ,[ short options ] );
```

The parameter *maxlen* specifies the maximum length in bytes of the character string pointed to by *name*, the actual length of which is returned by *namelen*. All three of these parameters must either be supplied or be absent.

Syntax for TAL Programmers

```
error := PROCESSSTRING_SCAN_ ( string:length       ! i:i
                              ,[ length-used ]     ! o
                              ,[ processhandle ]   ! o
                              ,[ stringtype ]      ! o
                              ,[ name:maxlen ]     ! o:i
                              ,[ namelen ]         ! o
                              ,[ cpu ]             ! o
                              ,[ pin ]             ! o
                              ,[ options ] );      ! i
```

Parameters

error returned value

INT

is a file-system error number indicating the outcome of the operation.

string:*length* input:input

STRING .EXT:ref:*, INT:value

is a character string to be searched to find a valid process string. *string* must be exactly *length* bytes long. A valid process string must begin at the first character of *string*. It can occupy the entire length of *string*, or it can occupy the left-hand portion and be followed by a character that is not valid in that part of a process string. If a node name is not present in the process string, the current default value in the =_DEFAULTS DEFINE is used for determining the process handle.

length-used output

INT .EXT:ref:1

if present, returns the number of characters in *string* that are part of the process string. If error 13 is returned, *length-used* is the number of characters that were accepted as valid before the name was determined to be invalid.

processhandle output

INT .EXT:ref:10

if present, returns the process handle of the designated process. A null process handle (-1 in each word) is returned if the designated process does not exist or if the form of the process string does not designate a particular process (for example, if only *cpu* is supplied).

stringtype output

INT .EXT:ref:1

if present, returns a value indicating the form of the process string contained in *string*, and therefore which output parameters have significant values. Valid values are:

0 Asterisk form (that is, "*")
1 Single processor form (for example, "2")
2 processor, PIN form (for example, "2,137")
3 Name form (for example, "$PSRV")

name:*maxlen* output:input

STRING .EXT:ref:*, INT:value

if present, returns the name of a node or process. If *stringtype* is less than 3, *name* returns the node name that was contained in the process string (if no node name was specified, the returned value of *namelen* is 0). If *stringtype* is 3, the returned value is the specified process name, including the node name, if present.

maxlen is the length in bytes of the string variable *name*.

namelen output

INT .EXT:ref:1

if present, returns the actual length of the value returned in *name*. If an error occurs, 0 is returned.

cpu output

INT .EXT:ref:1

if present, returns the processor value contained in the process string when *stringtype* is 1 or 2; otherwise -1 is returned.

pin output

INT .EXT:ref:1

if present, returns the PIN value contained in the process string when *stringtype* is 2; otherwise -1 is returned.

options input

INT:value

specifies desired options. The fields are:

<15>	0	Error 13 occurs if *options* <15> =0 and the input string name exceeds 6 characters including the '$' character.
	1	Causes an input string exceeding 6 characters to be accepted without error..
<0:14>		Reserved (specify 0)

When *options* is omitted, 0 is used.

Considerations

- A process string is a string of characters that identifies a process or a set of processes. Process strings are commonly used in command lines (for example, in the TACL STATUS command). PROCESSSTRING_SCAN_ accepts process strings in these forms:

```
[\node.]cpu,pin
[\node.]cpu
[\node.]$process-name
[\node.]*
```

- If you request the *processhandle*, PROCESSSTRING_SCAN_ verifies that the process exists. If the process does not exist, a null process handle (-1 in each word) is returned. If you supply a process name that represents an existing process pair, the returned process handle is that of the current primary.

PROCESSTIME Procedure (Superseded by PROCESS_GETINFOLIST_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The PROCESSTIME procedure returns the process execution time of any process in the network. Process time is the processor time in microseconds that the process has consumed; processor time used for Guardian procedures called is also included.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
process-time := PROCESSTIME ( [ cpu,pin ]        ! i
                            ,[ sysid ] );        ! i
```

Parameters

process-time returned value

FIXED

is the process execution time, in microseconds, of the specified process in the network.

-1F indicates that the process does not exist.

-2F indicates that the system is unavailable or does not exist; the procedure cannot get resources (link control blocks).

> 0F indicates that PROCESSTIME was successful.

cpu,pin input

INT:value

is the processor (in bits <4:7> with <0:3> not used) and PIN (in bits <8:15>) number of the process whose execution time is to be returned. If *cpu,pin* is omitted, the *cpu,pin* of the current process (calling process) is used, even if *sysid* is different than the current system.

sysid input

INT:value

is the system number. *sysid* defaults to the current system.

Considerations

You cannot use PROCESSTIME for a high-PIN process except when omitting *cpu,pin*. This is because a high-PIN cannot fit into *cpu,pin*.

Example

```
IF ( PROCESS^TIME := PROCESSTIME ( CPU^PIN , SYS^NUM )) >= 0F
   THEN ...    ! successful.
   ELSE ...    ! PROCESSTIME not available.
```

PROGRAMFILENAME Procedure (Superseded by PROCESS_GETINFOLIST_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The PROGRAMFILENAME procedure is used to obtain the name of the calling process's program file.

The main use of this procedure is to allow a primary process to create its backup process without having to hard code the program file name into the source program.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
CALL PROGRAMFILENAME ( program-file );              ! o
```

Parameters

program-file output

INT:ref:12

is an array where PROGRAMFILENAME returns the internal-format file name of the process's program file.

Example

```
CALL PROGRAMFILENAME ( MYPROG );
```

PURGE Procedure (Superseded by FILE_PURGE_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The PURGE procedure is used to delete a disk file that is not open. When PURGE is executed, the disk file name is deleted from the volume's directory, and any disk space previously allocated to that file is made available to other files.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
CALL PURGE ( file-name );          ! i
```

Parameters

file-name input

INT:ref:12

is an array containing the internal-format file name of the disk file to be purged. To purge either a permanent or temporary disk file, *file-name* must be of the form:

Permanent Disk File

[0:3] $*volname* (blank-fill)
or
sysnum volname (blank-fill)

[4:7] *subvol-name* (blank-fill)

[8:11] *file-id* (blank-fill)

Temporary Disk File

[0:3] $*volname* (blank-fill)
or
sysnum volname (blank-fill)

[4:11] #*temporary-file-id*

Condition Code Settings

< (CCL) indicates that the PURGE failed (call FILEINFO or FILE_GETINFO_). Note, however, that in the case of a disk free-space error (such as file-system errors 52, 54, 58), the file is purged, and an error returns.

= (CCE) indicates that the file purged successfully.

> (CCG) indicates that the device is not a disk.

Considerations

- Purge failure

  If PURGE fails, the reason for the failure can be determined by calling FILEINFO or FILE_GETINFO_, passing -1 as the *filenum* parameter.

- Purging a file audited by the Transaction Management Facility (TMF)

  If the file is a file audited by TMF and there are pending transaction-mode record locks or file locks, any attempt to purge that file fails with file error 12, whether or not openers of the file still exist.

When an audited file is purged, all corresponding dump records are deleted from the TMF catalog. If TMF is not active, attempts to purge an audited file fail with file-system error 82.

- Purging a partitioned file

  When you purge the primary partition of a partitioned file, the file system automatically purges all the other partitions located anywhere in the network that are marked as secondary partitions. A secondary partition is marked as such if it created at the same time as the primary partition.

- Security consideration

  File purging normally is performed in a logical fashion; the data is not necessarily overwritten or erased, but rather pointers are changed to show the data to be absent. For security reasons, you might want to set the CLEARONPURGE flag for a file, using either function 1 of the SETMODE procedure or the File Utility Program SECURE command. Either way, this option causes all data to be physically erased (overwritten with zeros) when the file is purged.

- Expiration dates

  PURGE checks the expiration date of a file before it purges the file. If the expiration date is later than the current date, PURGE does not purge the file and returns error code 1091.

Safeguard Considerations

For information on files protected by Safeguard, see the *Safeguard Reference Manual*.

OSS Considerations

This procedure operates only on Guardian objects. If an OSS file is specified, error 1163 occurs.

Example

```
CALL PURGE ( OLD^FILE^NAME );
```

PUTPOOL Procedure (Superseded by POOL_* Procedures)

BOOKMARK

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development. *POOL procedures are replaced by POOL_* procedures. There is no one-for-one replacement.

The PUTPOOL procedure returns a block of memory to a buffer pool.

Syntax for C Programmers

```
#include <cextdecs(PUTPOOL)>

_cc_status PUTPOOL( short *pool-head
                   ,char *pool-block );
```

The function value returned by PUTPOOL, which indicates the condition code, can be interpreted by the `_status_lt()`, `_status_eq()`, or `_status_gt()` function (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL PUTPOOL ( pool-head                    ! i,o
              ,pool-block );                ! i
```

Parameters

pool-head input, output

INT .EXT:ref:19

is the address of the pool head of the pool from which the block of memory was obtained using GETPOOL.

pool-block input

STRING .EXT:ref:*

is the address of the block to be returned to the pool.

Condition Code Settings

< (CCL) indicates that the data structures are invalid or that *pool-block* is not a block in the buffer pool.

= (CCE) indicates that the operation is successful.

> (CCG) is not returned from PUTPOOL.

Considerations

GETPOOL and PUTPOOL do not check pool data structures on each call. A process that destroys data structures can fail on a call to GETPOOL or PUTPOOL: a Guardian TNS process can get an instruction failure trap (trap 1) or an invalid address reference trap (trap 0); an OSS or native process can receive a SIGILL or SIGSEGV signal.

Example

```
CALL PUTPOOL ( pool^head, pblock );
```

`pool^head` is the pool head of the pool from which the block of memory was obtained, and `PBLOCK` is the block to be returned to the pool.

Related Programming Manual

For programming information about the PUTPOOL memory-management procedure, see the *Guardian Programmer's Guide*.

13 Guardian Procedure Calls (R)

This section contains detailed reference information for all user-accessible Guardian procedure calls beginning with the letter R. Table 13-1 lists all the procedures in this section.

Table 13-1. Procedures Beginning With the Letter R

RAISE_ Procedure

Note. This procedure can be called only from native processes.

RAISE_ is the pTAL procedure name for the C `raise()` function. The C `raise()` function complies with the POSIX.1 standard.

See the $SYSTEM.SYSTEM.HSIGNAL header file for the pTAL prototype definitions. For a discussion of each parameter and other procedure considerations, see the `raise(3)` function reference page either online or in the *Open System Services System Calls Reference Manual*.

Considerations

When RAISE_ is used to stop a process, the operating system supplies a completion code in the system message and, for OSS processes, in the OSS process termination status as follows:

- If the signal is handled by SIG_DFL, a completion code of 9 is returned and the signal number is returned in the termination information.
- If the signal is handled by the default CRE signal handler, a completion code of 3 is returned with 0 in the termination information.

For a list of completion codes, see Appendix C, Completion Codes.

READ[X] Procedures

Summary

The READ[X] procedures return data from an open file to the application process's data area. READ is intended for use with 16-bit addresses, while READX is intended for use with 32-bit extended addresses. Therefore, the data buffer for READX can be either in the caller's stack segment or any extended data segment.

The READ[X] procedures sequentially read a disk file. For key-sequenced, relative, and entry-sequenced files, the READ[X] procedures read a subset of records in the

file. (A subset of records is defined by an access path, positioning mode, and comparison length.)

Syntax for C Programmers

```
#include <cextdecs(READ)>

_cc_status READ ( short filenum
                 ,short _near *buffer
                 ,unsigned short read-count
                 ,[unsigned short _near *count-read ]
                 ,[ __int32_t tag ] );


#include <cextdecs(READX)>

_cc_status READX ( short filenum
                  ,char _far *buffer
                  ,unsigned short read-count
                  ,[unsigned short _far *count-read ]
                  ,[ __int32_t tag ] );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.
- The function value returned by READ[X], which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL READ[X] ( filenum                    ! i
              ,buffer                     ! o
              ,read-count                 ! i
              ,[ count-read ]             ! o
              ,[ tag ] );                 ! i
```

Parameters

filenum input

INT:value (Use with both READ and READX)

is the number of an open file that identifies the file to be read.

buffer output

INT:ref:* (Use with READ)
STRING .EXT:ref:* (Use with READX)

is an array in the application process in which the information read from the file is returned. The *buffer* for READ can be only in the user's stack area, while the *buffer* for READX can be in the caller's stack segment or in any extended data segment.

read-count input

INT:value (Use with both READ and READX)

is the number of bytes to be read:

{0:57344}	for disk files (see Disk File Considerations on page 13-7 and Appendix J, System Limits)
{0:32755}	for terminal files
{0:57344}	for other nondisk files (device dependent)
{0:57344}	for $RECEIVE and process files
{0:80}	for the operator console

count-read output

INT:ref:1 (Use with READ)
INT .EXT:ref:1 (Use with READX)

is for waited I/O only. It returns a count of the number of bytes returned from the file into *buffer*.

tag input

INT(32):value (Use with both READ and READX)

is for nowait I/O only. *tag* is a value you define that uniquely identifies the operation associated with this READ[X].

Note. The system stores the *tag* value until the I/O operation completes. The system then returns the *tag* information to the program in the *tag* parameter of the call to AWAITIO[X], thus indicating that the operation completed. If READX is used, the user must call the AWAITIOX procedure to complete the I/O. If READ is used, the user may use either AWAITIO or AWAITIOX to complete the I/O.

Condition Code Settings

< (CCL)	indicates that an error occurred (call FILE_GETINFO_ or FILEINFO).
< (CCL)	is also returned following a successful read with an insertion-ordered alternate key path if the alternate key value of the current record is equal to the alternate key value in this record along that path. A call to FILE_GETINFO_ or FILEINFO shows that error 551 occurred; this error is advisory only and does not indicate an unsuccessful read operation.
= (CCE)	indicates that the READ[X] is successful.
> (CCG)	for disk and nondisk devices, indicates that the end of file (EOF) is encountered (no more records in this subset); for the $RECEIVE file, a system message is received (call FILE_GETINFO_ or FILEINFO).

Considerations

- READ versus READX

  Use READ when the buffer has a 16-bit address, and use READX when the buffer has a 32-bit extended address. Therefore, the data buffer for READX can be either in the caller's stack segment or any extended data segment.

- Waited READ[X]

  If a waited READ[X] is executed, the *count-read* parameter indicates the number of bytes actually read.

- Nowait READ[X]

  If a nowait READ[X] is executed, *count-read* has no meaning and can be omitted. The count of the number of bytes read is obtained through the *count-transferred* parameter of the AWAITIO[X] procedures when the I/O operation completes.

  The READ[X] procedure must complete with a call to the AWAITIO[X] procedure when it is used with a file that is opened nowait. If READX is used, you must call AWAITIOX to complete the I/O. If READ is used, you may use either AWAITIO or AWAITIOX to complete the I/O.

▲ **WARNING.** When using nowait file I/O, data corruption might occur if the READ buffer is modified before the AWAITIOX that completes the call.

  It is possible to initiate concurrent nowait read operations that share the same data buffer. To do this successfully with files opened by FILE_OPEN_, you must use SETMODE function 72 to cause the system to use an intermediate buffer in the process file segment (PFS) for I/O transfers. With files opened by OPEN, a PFS buffer is used by default.

- READ[X] from process files

  The action for a READ of a process file is the same as that for a WRITEREAD with zero *write-count*.

- READ[X] call when default locking mode is in effect

  If the default locking mode is in effect when a call to READ[X] is made to a locked file, but the *filenum* of the locked file differs from the *filenum* in the call, the caller of READ[X] is suspended and queued in the "locking" queue behind other processes attempting to lock or read the file or record.

Note. A deadlock condition occurs if a call to READ[X] is made by a process having multiple opens on the same file and the *filenum* used to lock the file differs from the *filenum* supplied to READ[X].

- Read call when alternate locking mode is in effect

If the alternate locking mode is in effect when READ[X] is called, and the file or record is locked through a file number other than that supplied in the call, the call is rejected with file-system error 73 (file is locked).

- Locking mode for read

  The locking mode is specified by the SETMODE procedure, function 4. If you encounter error 73 (file is locked), you do not need to call SETMODE for every READ[X]. SETMODE stays in effect indefinitely (for example, until another SETMODE is performed or the file is closed), and there is no additional overhead involved.

- Considerations for READX only

  - *buffer* and *count-transferred* can be in the user stack or in an extended data segment. *buffer* and *count-transferred* cannot be in the user code space.
  - The *buffer* address and *count-transferred* address must be relative; they cannot be an absolute extended address.
  - If *buffer* or *count-transferred* is in a selectable extended data segment, the segment must be in use at the time of the call. Flat segments allocated by a process are always accessible to the process.
  - The size of the transfer is subject to current restrictions for the type of file.
  - If the file is opened for nowait I/O, and the buffer is in an extended data segment, you must not deallocate or reduce the size of the extended data segment before the I/O finishes with a call to AWAITIOX or is canceled by a call to CANCEL or CANCELREQ.
  - If the file is opened for nowait I/O, you must not modify the buffer before the I/O finishes with a call to AWAITIOX. This also applies to other processes that might be sharing the segment. It is the application's responsibility to ensure this.
  - If the file is opened for nowait I/O and the I/O has been initiated with these routines, the I/O must be completed with a call to AWAITIOX (not AWAITIO).
  - If the file is opened for nowait I/O, a selectable extended data segment containing the buffer need not be in use at the time of the call to AWAITIOX.
  - Nowait I/O initiated with these routines can be canceled with a call to CANCEL or CANCELREQ. The I/O is canceled if the file is closed before the I/O finishes or AWAITIOX is called with a positive time limit and specific file number, and the request times out.
  - A file opened by FILE_OPEN_ uses direct I/O transfers by default; you can use SETMODE 72 to force the system to use an intermediate buffer in the process file segment (PFS) for I/O transfers. A file opened by OPEN uses a PFS buffer for I/O transfers, except for large transfers to DP2 disks.

  - If the extended address of the buffer is odd, bounds checking rounds the address to the next lower word boundary and checks an extra byte as well. The odd address is used for the transfer.

- Queue files

  READ[X] can be used to perform a nondestructive read of a queue file record. If KEYPOSITION[X] is used to position to the beginning of the file, the first READ[X] performed returns a record with a length of 8 bytes and contents of all zeroes. Subsequent READ[X] calls will return data from records written to the file.

Disk File Considerations

- Large data transfers for unstructured files using default mode

  For the read procedures (READ[UPDATE] [LOCK] [X]), using default mode allows I/O sizes for unstructured files to be as large as 56 kilobytes (57,344), if the unstructured buffer size is 4 KB (4096). Default mode here refers to the mode of the file if SETMODE function 141 is not invoked.

  For an unstructured file with an unstructured buffer size other than 4 KB, DP2 automatically adjusts the unstructured buffer size to 4 KB, if possible, when an I/O larger than 4KB is attempted. However, this adjustment is not possible for files that have extents with an odd number of pages; in such cases an I/O over 4 KB is not possible. Note that the switch to a different unstructured buffer size will have a transient performance impact, so it is recommended that the size be initially set to 4 KB, which is the default. Transfer sizes over 4 KB are not supported in default mode for unstructured access to structured files.

- Large data transfers using SETMODE 141

  For READX only, large data transfers (more than 4096 bytes) can be done for unstructured access to structured or unstructured files, regardless of unstructured buffer size, by using SETMODE function 141. When SETMODE 141 is used to enable large data transfers, it is permitted to specify up to 56K (57344) bytes for the *read-count* parameter. See Table 14-4 on page 14-63 for use of SETMODE function 141.

- Structured files

  - a subset of records for sequential READ[X]s

    The subset of records read by a series of calls to READ[X] is specified through the POSITION or KEYPOSITION procedures.

  - reading of an approximate subset of records

    If an approximate subset is being read, the first record returned is the one whose key field, as indicated by the current key specifier, contains a value equal to or greater than the current key. Subsequent reading of the subset returns successive records until the last record in the file is read (an EOF indication is then returned).

- reading of a generic subset of records

  If a generic subset is being read, the first record returned is the one whose key field, as designated by the current-key specifier, contains a value equal to the current key for *comparison-length* bytes. Subsequent reading of the file returns successive records whose key matches the current key (for *comparison-length* bytes). When the current key no longer matches, an EOF indication returns.

  For relative and entry-sequenced files, a generic subset of the primary key is equivalent to an exact subset.

- reading of an exact subset of records

  If an exact subset is being read, the only records returned are those whose key field, as designated by the current-key specifier, contains a value of exactly the comparison length bytes (see KEYPOSITION[X] Procedures (Superseded by FILE_SETKEY_ Procedure)) and is equal to the key. When the current key no longer matches, an EOF indication returns. The exact subset for a key field having a unique value is at most one record.

- indicators after READ[X]

  After a successful READ[X], the current-state indicators have these values:

Current position	record just read
Positioning mode	unchanged
Comparison length	unchanged
Current primary-key value	set to the value of the primary-key field in the record

- Read-reverse action on current and next record pointers

  Following a call to READ when reverse-positioning mode is in effect, the *next-record-pointer* contains the record number or address which precedes the current record number or address.

  Following a read of the first record in a file (where *current-record-pointer* = 0) with reverse positioning, the *next-record-pointer* will contain an invalid record number or address since no previous record exists. A subsequent call to READ would return an “end-of-file” error, whereas a call to WRITE would return an “illegal position” error (error 550) since an attempt was made to write beyond the beginning of the file.

- Unstructured files

  - READ[X]s

    Data transfer begins from an unstructured disk file at the position indicated by the next-record pointer.

    The READ[X] procedure reads records sequentially on the basis of a beginning relative byte address (RBA) and the length of the records read.

- odd unstructured

  If the unstructured file is created with the odd unstructured attribute (also known as ODDUNSTR) set, the number of bytes read is exactly the number of bytes specified with *read-count*. If the odd unstructured attribute is not set when the file is created, the value of *read-count* is rounded up to an even number before the READ[X] is executed.

  You set the odd unstructured attribute with the FILE_CREATE_, FILE_CREATELIST_, or CREATE procedure, or with the File Utility Program (FUP) SET and CREATE commands.

- READ[X] count

  Unstructured files are transparently blocked. The BUFFERSIZE file attribute value, if not set by the user, defaults to 4096 bytes. The BUFFERSIZE attribute value (which is set by specifying SETMODE function 93) does not constrain the allowable *read-count* in any way. However, there is a performance penalty if the READ[X] does not start on a BUFFERSIZE boundary and does not have a *read-count* that is an integral multiple of the BUFFERSIZE. The DP2 disk process executes your requested I/O in (possibly multiple) units of BUFFERSIZE blocks starting on a block boundary.

- of *count-read* for unstructured READ[X]s

  After a successful call to READ[X] for an unstructured file, the value returned in *count-read* is determined by:

```
count-read := $MIN(read-count &
                   eof-pointer - next-record pointer)
```

- pointers after READ[X]

  After a successful READ[X] to an unstructured file, the file pointers are:

  CCG = 1 if the next-record pointer = EOF pointer; otherwise CCG = 0

  current-record pointer = old next-record pointer

  next-record pointer = old next-record pointer + *count-read*

Errors for READX Only

In addition to the errors currently returned from READ, error 22 is returned from READX when:

- The address of a parameter refers to the selectable segment area but no selectable segment is in use at the time of the call.
- The address of a parameter is extended, but it is an absolute address and the caller is not privileged.

Example

```
CALL READ ( FILE^NUM , IN^BUFFER , 72 , NUM^XFERRED );

! The READ permits up to 72 bytes to be read into IN^BUFFER,
! and the count actually read returns into NUM^XFERRED.
```

Related Programming Manuals

For programming information about the READ file-system procedure, see the *Guardian Programmer's Guide*, the *Enscribe Programmer's Guide*, and the data communication manuals.

READ^FILE Procedure

Summary

The READ^FILE procedure is used to read a file sequentially. The file must be open with read or read/write access.

READ^FILE is a sequential I/O (SIO) procedure and should be used only with files that have been opened by OPEN^FILE.

Syntax for C Programmers

```
#include <cextdecs(READ_FILE )>

short READ_FILE ( short _near *file-fcb
                 ,short _near *buffer
                 ,[ short _near *count-returned ]
                 ,[ short prompt-count ]
                 ,[ short max-read-count ]
                 ,[ short nowait ] );
```

Syntax for TAL Programmers

```
error := READ^FILE ( file-fcb                ! i
                    ,buffer                  ! o
                    ,[ count-returned ]      ! o
                    ,[ prompt-count ]        ! i
                    ,[ max-read-count ]      ! i
                    ,[ nowait ] );           ! i
```

Parameters

error returned value

INT

returns a file-system or SIO procedure error indicating the outcome of the read.

If abort-on-error mode is in effect, the only possible values for *error* are:

0 No error

1 End of file

6 System message (only if user requested system messages through SET^SYSTEMMESSAGES or SET^SYSTEMMESSAGESMANY)

111 Operation aborted because of BREAK (if BREAK is enabled). If *nowait* is not zero, and if abort-on-error is in effect, the only possible value for *error* is 0.

file-fcb input

INT:ref:*

identifies the file to be read.

buffer output

INT:ref:*

is where the data is returned. The buffer must be located within 'G'[0:32767] process data area.

count-returned output

INT:ref:1

returns the number of bytes returned to *buffer*. If I/O is *nowait*, this parameter has no meaning and can be omitted. The count is then obtained in the call to WAIT^FILE.

prompt-count input

INT:value

is a count of the number of bytes in *buffer*, starting with element zero, to be used as an interactive prompt for terminals or interprocess files. If omitted, the interactive prompt character defined in OPEN^FILE is used.

max-read-count input

INT:value

specifies the maximum number of bytes to be returned to *buffer*. If omitted or if it exceeds the file's logical record length, the logical record length is used for this file.

nowait input

INT:value

indicates whether or not to wait for the I/O operation to complete in this call. If omitted or zero, then "wait" is indicated. If not zero, the I/O operation must be completed in a call to WAIT^FILE.

Considerations

- Terminal or Process File

  If the file is a terminal or process, a WRITEREAD operation is performed using the interactive prompt character or *prompt-count* character from *buffer*. For $RECEIVE, READ^FILE does a READUPDATE instead of a READ.

Example

```
ERROR := READ^FILE ( IN^FILE , BUFFER , COUNT );
```

Related Programming Manual

For programming information about the READ^FILE procedure, see the *Guardian Programmer's Guide*.

READEDIT Procedure

Summary

The READEDIT procedure reads one line from a specified EDIT file and returns it to the caller in unpacked format.

READEDIT is an IOEdit procedure and can only be used with files that have been opened by OPENEDIT or OPENEDIT_.

Syntax for C Programmers

Note. In the TNS/E environment the CEXTDECS file uses the *int* data type for 32-bit values. This is a change from the TNS and TNS/R environments where CEXTDECS uses the *long* data type for 32-bit values.

```
#include <cextdecs(READEDIT)>

short READEDIT ( short filenum
                ,[ __int32_t *record-number ]
                ,char *unpacked-line
                ,short unpacked-limit
                ,short *unpacked-length
                ,[ reserved parameter ]
                ,short prompt-count
                ,[ short spacefill ]
                ,[ short full-length ] );
```

- The sixth parameter to READEDIT is reserved for internal use. You must supply a placeholder comma for this parameter when using the parameters that follow it.

Syntax for TAL Programmers

```
error := READEDIT ( filenum                    ! i
                   ,[ record-number ]          ! i,o
                   ,unpacked-line              ! o
                   ,unpacked-limit             ! i
                   ,unpacked-length            ! o
                   ,[ reserved parameter ]
                   ,[ spacefill ]              ! i
                   ,[ full-length ] );         ! i
```

Parameters

error returned value

INT

returns a file-system error number indicating the outcome of the operation.

filenum input

INT:value

specifies the file number of the open file that is to be read.

record-number input, output

INT(32):ref:1

if present, specifies the record number of the line to be read. If *record-number*:

- is greater than or equal to 0, READEDIT reads the line (if any) with the smallest EDIT line number that is greater than or equal to the value of 1000 times *record-number*.
- is -1, READEDIT reads the lowest-numbered line (if any) in the file.
- is -2, READEDIT reads the highest-numbered line (if any) in the file.
- is -3 or omitted, READEDIT reads the line (if any) with the smallest record number that is greater than or equal to the next record number.

record-number returns the value of the file's new current record number after the read has been performed. This is equal to the record number of the line that was read, or it is -2 (end of file) if no line was read.

unpacked-line output

STRING .EXT:ref:*

is a string array that contains the line in unpacked format that is the outcome of the operation. The length of the unpacked line is returned in the *unpacked-length* parameter.

unpacked-limit input

INT:value

specifies the length in bytes of the string variable *unpacked-line*.

unpacked-length output

INT .EXT:ref:1

returns the actual length in bytes of the value returned in *unpacked-line*. If *unpacked-line* is not large enough to contain the value that is the output of the operation, *unpacked-length* returns a negative value.

[reserved parameter]

is reserved for internal use. When using the parameters that follow, you must supply a placeholder comma for the reserved parameter.

spacefill input

INT:value

if present and not equal to 0, specifies that if the value returned in *unpacked-line* is shorter than *unpacked-limit*, READEDIT should fill the unused part of *unpacked-line* with space characters. Otherwise, READEDIT does nothing to the unused part of *unpacked-line*.

full-length input

INT:value

if present and not equal to 0, specifies that all trailing space characters (if any) in the line being processed should be retained in the output line and should be counted in the value returned in *unpacked-length*. Otherwise, trailing space characters are discarded and not counted in *unpacked-length*.

Example

```
error := READEDIT (filenumber, record^num, line^image,
                   line^maxlength, line^actual^length, ,
                   space^fill);
IF error THEN ...  ! handle error
IF line^actual^length < 0 THEN ... ! buffer (line^image)
                                   !  too small for return
                                   !    value
```

Related Programming Manual

For programming information about the READEDIT procedure, see the *Guardian Programmer's Guide*.

READEDITP Procedure

Summary

The READEDITP procedure reads one line from a specified EDIT file and returns it to the caller in EDIT packed line format.

READEDITP is an IOEdit procedure and can only be used with files that have been opened by OPENEDIT or OPENEDIT_.

Syntax for C Programmers

Note. In the TNS/E environment the CEXTDECS file uses the *int* data type for 32-bit values. This is a change from the TNS and TNS/R environments where CEXTDECS uses the *long* data type for 32-bit values.

```
#include <cextdecs(READEDITP)>

short READEDITP ( short filenum
                 ,[ __int32_t *record-number ]
                 ,char *packed-line
                 ,short packed-limit
                 ,short *packed-length );
```

Syntax for TAL Programmers

```
error := READEDITP ( filenum                    ! i
                    ,[ record-number ]          ! i,o
                    ,packed-line                ! o
                    ,packed-limit               ! i
                    ,packed-length );           ! o
```

Parameters

error returned value

INT

returns a file-system error number indicating the outcome of the operation.

filenum input

INT:value

specifies the file number of the open file that is to be read.

record-number input, output

INT(32):ref:1

if present, specifies the record number of the line to be read. If *record-number*:

- is greater than or equal to 0, READEDITP reads the line (if any) with the smallest EDIT line number that is greater than or equal to the value of 1000 times *record-number*.
- is -1, READEDITP reads the lowest-numbered line (if any) in the file.
- is -2, READEDITP reads the highest-numbered line (if any) in the file.

- is -3 or omitted, READEDITP reads the line (if any) with the smallest record number that is greater than or equal to the next record number.

record-number returns the value of the file's new current record number after the read has been performed. This is equal to the record number of the line that was read, or it is -2 (end of file) if no line was read.

packed-line output

STRING .EXT:ref:*

is a string array that contains the line in unpacked format that is the outcome of the operation. The length of the unpacked line is returned in the *packed-length* parameter.

packed-limit input

INT:value

specifies the length in bytes of the string variable *packed-line*.

packed-length output

INT .EXT:ref:1

returns the actual length in bytes of the value returned in *packed-line*. If *packed-line* is not large enough to contain the value that is the output of the operation, *packed-length* returns a negative value.

Example

```
error := READEDITP ( filenumber, record^num, line^image,
                     line^maxlength, line^actual^length );
IF error THEN ...   ! handle error
IF line^actual^length < 0 THEN ... ! buffer (line^image)
                                   !  too small for return
                                   !    value
```

Related Programming Manual

For programming information about the READEDITP procedure, see the *Guardian Programmer's Guide*.

READLOCK[X] Procedures

Summary

The READLOCK and READLOCKX procedures sequentially lock and read records in a disk file, exactly like the combination of LOCKREC and READ[X]. READLOCK is intended for use with 16-bit addresses, while READLOCKX is intended for use with 32-bit extended addresses. Therefore, the data buffer for READLOCKX can be either in the caller's stack segment or any extended data segment.

Syntax for C Programmers

```
#include <cextdecs(READLOCK)>

_cc_status READLOCK ( short filenum
                     ,short _near *buffer
                     ,unsigned short read-count
                     ,[unsigned short _near *count-read ]
                     ,[ __int32_t tag ] );


#include <cextdecs(READLOCKX)>

_cc_status READLOCKX ( short filenum
                      ,char _far *buffer
                      ,unsigned short read-count
                      ,[unsigned short _far *count-read ]
                      ,[ __int32_t tag ] );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.
- The function value returned by READLOCK[X], which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL READLOCK[X] ( filenum                          ! i
                  ,buffer                           ! o
                  ,read-count                       ! i
                  ,[ count-read ]                   ! o
                  ,[ tag ] );                       ! i
```

Parameters

filenum input

INT:value (Use with both READLOCK and READLOCKX)

is the number of an open file that identifies the file to be read.

buffer output

INT:ref:* (Use with READLOCK)
STRING .EXT:ref:* (Use with READLOCKX)

is an array in the application process where the information read from the file returns.

read-count input

INT:value (Use with both READLOCK and READLOCKX)

is the number of bytes to be read: {0:4096}.

count-read output

INT:ref:1 (Use with READLOCK)
INT .EXT:ref:1 (Use with READLOCKX)

is for wait I/O only. *count-read* returns a count of the number of bytes returned from the file into *buffer*.

tag input

INT(32):value (Use with both READLOCK and READLOCKX)

is for nowait I/O only. *tag* is a value you define that uniquely identifies the operation associated with this READLOCK[X].

Note. The system stores the *tag* value until the I/O operation completes. The system then returns the *tag* information to the program in the *tag* parameter of the call to AWAITIO[X], thus indicating that the operation completed. If READLOCKX is used, you must call AWAITIOX to complete the I/O. If READLOCK is used, you may use either AWAITIO or AWAITIOX to complete the I/O.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILE_GETINFO_ or FILEINFO).

< (CCL) is also returned following a successful read with an insertion-ordered alternate key path if the alternate key value of the current record is equal to the alternate key value in this record along that path. A call to FILE_GETINFO_ or FILEINFO shows that an error 551 occurred; this error is advisory only and does not indicate an unsuccessful read operation.

= (CCE) indicates that the READLOCK[X] is successful.

> (CCG) indicates end of file (EOF). There are no more records in this subset.

Considerations

- READLOCK versus READLOCKX

  Use READLOCK if the buffer has a 16-bit address and use READLOCKX if the buffer has a 32-bit extended address. Therefore, the data buffer for READLOCKX can be either in the caller's stack segment or any extended data segment.

- Nowait I/O and READLOCK[X]

  If the READLOCK[X] procedure is used to initiate an operation with a file-opened nowait, it must complete with a corresponding call to the AWAITIO[X] procedure. If READLOCKX is used, you must call AWAITIOX to complete the I/O. If READLOCK is used, you may use either AWAITIO or AWAITIOX to complete the I/O.

▲ **WARNING.** When using nowait file I/O, data corruption might occur if the READ buffer is modified before the AWAITIOX that completes the call.

- READLOCK[X] for key-sequenced, relative, and entry-sequenced files

  For key-sequenced, relative, and entry-sequenced files, a subset of the file (defined by the current access path, positioning mode, and comparison length) is locked and read with successive calls to READLOCK[X].

  For key-sequenced, relative, and entry-sequenced files, the first call to READLOCK[X] after a positioning (or open) locks and then returns the first record of the subset. Subsequent calls to READLOCK[X] without intermediate positioning locks, returns successive records in the subset. After each of the subset's records are read, the position of the record just read becomes the file's current position. An attempt to read a record following the last record in a subset returns an EOF indication.

- Locking records in an unstructured file

  READLOCK[X] can be used to lock record positions, represented by a relative byte address (RBA), in an unstructured file. When sequentially reading an unstructured file with READLOCK[X], each call to READLOCK[X] first locks the RBA stored in the current next-record pointer and then returns record data beginning at that

pointer for *read-count* bytes. After a successful READLOCK[X], the current-record pointer is set to the previous next-record pointer, and the next-record pointer is set to the previous next-record pointer plus *read-count*. This process repeats for each subsequent call to READLOCK[X].

- See Considerations on page 13-5.

Considerations for READLOCKX Only

- The buffer and count transferred may be in the user stack or in an extended data segment. The buffer and count transferred cannot be in the user code space.
- If the buffer or count transferred is in a selectable extended data segment, the segment must be in use at the time of the call. Flat segments allocated by a process are always accessible to the process.
- The size of the transfer is subject to current restrictions for the type of file.
- If the file is opened for nowait I/O, and the buffer is in an extended data segment, you cannot deallocate or reduce the size of the extended data segment before the I/O completes with a call to AWAITIOX or is canceled by a call to CANCEL or CANCELREQ.
- If the file is opened for nowait I/O, you must not modify the buffer before the I/O completes with a call to AWAITIOX. This also applies to other processes that may be sharing the segment. It is the application's responsibility to ensure this.
- If the file is opened for nowait I/O, and the I/O has been initiated with these routines, the I/O must be completed with a call to AWAITIOX (not AWAITIO).
- If the file is opened for nowait I/O, a selectable extended data segment containing the buffer need not be in use at the time of the call to AWAITIOX.
- Nowait I/O initiated with these routines may be canceled with a call to CANCEL or CANCELREQ. The I/O is canceled if the file is closed before the I/O completes or AWAITIOX is called with a positive time limit and specific file number and the request times out.
- A file opened by FILE_OPEN_ uses direct I/O transfers by default; you can use SETMODE 72 to force the system to use an intermediate buffer in the process file segment (PFS) for I/O transfers. A file opened by OPEN uses a PFS buffer for I/O transfers, except for large transfers to DP2 disks.
- If the extended address of the buffer is odd, bounds checking rounds the address to the next lower word boundary and checks an extra byte as well. The odd address is used for the transfer.

Errors for READLOCKX Only

In addition to the errors currently returned from READLOCK, error 22 is returned from READLOCKX when:

- The address of a parameter is extended, but either the extended data segment is invalid or the address is for a selectable segment that is not in use at the time of the call.
- The address of a parameter is extended, but it is an absolute address and the caller is not privileged.

OSS Considerations

- This procedure operates only on Guardian objects. If an OSS file is specified, error 2 is returned.

Related Programming Manuals

For programming information about the READLOCK file-system procedure, see the *Enscribe Programmer's Guide*.

READUPDATE[X|XL] Procedures

Summary

The READUPDATE[X|XL] procedures read data from a disk or process file in anticipation of a subsequent write to the file. READUPDATE is intended for use with 16-bit addresses, READUPDATEX[L] is intended for use with 32-bit extended addresses. Therefore, the data buffer for READUPDATEX or READUPDATEXL can be either in the caller's stack segment or any extended data segment.

- Disk files

  READUPDATE[X|XL] is used for random processing. Data is read from the file at the position of the current-record pointer. A call to this procedure typically follows a corresponding call to POSITION or KEYPOSITION. The values of the current- and next-record pointers do not change with the call to READUPDATE[X|XL].

- Queue Files

  READUPDATE[X|XL] is not supported on queue files. An attempt to use READUPDATE[X|XL] will be rejected with error 2.

- Interprocess communication

  READUPDATE[X|XL] reads a message from the $RECEIVE file that is answered in a later call to REPLY[X|XL]. Each message read by READUPDATE[X|XL] must be replied to in a corresponding call to REPLY[X|XL].

Syntax for C Programmers

```
#include <cextdecs(READUPDATE)>

_cc_status READUPDATE ( short filenum
                       ,short _near *buffer
                       ,unsigned short read-count
                       ,[unsigned short _near *count-read ]
                       ,[ __int32_t tag ] );


#include <cextdecs(READUPDATEX)>

_cc_status READUPDATEX ( short filenum
                        ,char _far *buffer
                        ,unsigned short read-count
                        ,[unsigned short _far *count-read ]
                        ,[ __int32_t tag ] );


#include <cextdecs(READUPDATEXL)>

short READUPDATEXL ( short filenum
                    ,char _far *buffer
                    ,__int32_t read-count
                    ,[ __int32_t _far *count-read ]
                    ,[ long long tag ] );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

- The function value returned by READUPDATE[X], which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL READUPDATE[X] ( filenum                          ! i
                    ,buffer                           ! o
                    ,read-count                       ! i
                    ,[ count-read ]                   ! o
                    ,[ tag ] );                       ! i


error:= READUPDATEXL ( filenum                         ! i
                    ,buffer                           ! o
                    ,read-count                       ! i
                    ,[ count-read ]                   ! o
                    ,[ tag ] );                       ! i
```

Parameters

error returned value

INT (Use with READUPDATEXL)

is a file-system error number indicating the outcome of the operation.

0 (FEOK)

indicates a successful operation.

6 (FESYSMESSAGE)

indicates a successful operation that reads a system message. Valid only if filenum is $RECEIVE.

filenum input

INT:value (Use with READUPDATE, READUPDATEX, and READUPDATEXL)

is the number of an open file that identifies the file to be read.

buffer output

INT:ref:* (Use with READUPDATE)
STRING .EXT:ref:* (Use with READUPDATEX and READUPDATEXL)

is an array where the information read from the file returns.

read-count input

INT:value (Use with READUPDATE and READUPDATEX)
INT(32):value (Use with READUPDATEXL)

is the number of bytes to be read.

{0:4096} for disk files (see Disk File Considerations on page 13-7)
{0:57344} for $RECEIVE (Use with READUPDATE and READUPDATEX)
{0:2097152} for $RECEIVE (Use with READUPDATEXL)

count-read output

INT:ref:1 (Use with READUPDATE)
INT .EXT:ref:1 (Use with READUPDATEX)
INT(32).EXT:ref:1 (Use with READUPDATEXL)

is for wait I/O only. *count-read* returns a count of the number of bytes returned from the file into *buffer*.

tag input

INT(32):value (Use with READUPDATE and READUPDATEX)
INT(64):value (Use with READUPDATEXL)

is for nowait I/O only. *tag* is a value you define that uniquely identifies the operation associated with this READUPDATE[X|XL]. If the completed I/O operation has a 32-bit tag, the 64-bit tag is in the sign-extended value of the 32-bit *tag*.

Note. The system stores the *tag* value until the I/O operation completes. The system then returns the *tag* information to the program in the *tag* parameter of the call to AWAITIO[X|XL], thus indicating that the operation completed. If READUPDATEX or READUPDATEXL is used, you must call AWAITIOX or AWAITIOXL to complete the I/O. If READUPDATE is used, you may use either AWAITIO or AWAITIOX or AWAITIOXL to complete the I/O.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILE_GETINFO_ or FILEINFO, with *filenum* of 0).

< (CCL) is also returned following a successful read with an insertion-ordered alternate key path if the alternate key value of the current record is equal to the alternate key value in this record along that path. A call to FILE_GETINFO_ or FILEINFO shows that an error 551 occurred; this error is advisory only and does not indicate an unsuccessful read operation.

= (CCE) indicates that the READUPDATE[X] is successful.

> (CCG) indicates that a system message is received through $RECEIVE. (CCG is not returned by READUPDATE[X] for disk files.)

Considerations

- READUPDATE versus READUPDATEX[L]

  Use READUPDATE when the buffer has a 16-bit address; use READUPDATEX[L] when the buffer has a 32-bit extended address. Therefore, the data buffer for READUPDATEX[L] can be either in the caller's stack segment or any extended data segment.

- Random processing and positioning

  A call to READUPDATE[X|XL] returns the record from the current position in the file. Because READUPDATE[X|XL] is designed for random processing, it cannot be used for successive positioning through a subset of records as the READ[X] procedure does. Rather, READUPDATE[X|XL] reads a record after a call to POSITION or KEYPOSITION, possibly in anticipation of a subsequent update through a call to the WRITEUPDATE[X] procedure.

- Calling READUPDATE[X|XL] after READ[X]

  A call to READUPDATE[X|XL] after a call to READ[X], without intermediate positioning, returns the same record as the call to READ[X].

- Waited READUPDATE[X|XL]

  If a waited READUPDATE[X|XL] is executed, the *count-read* parameter indicates the number of bytes actually read.

- Nowait I/O and READUPDATE[X|XL]

  If a nowait READUPDATE[X|XL] is executed, *count-read* has no meaning and can be omitted. The count of the number of bytes read is obtained when the I/O operation completes through the *count-transferred* parameter of the AWAITIO[X|XL] procedure.

  The READUPDATE[X|XL] procedure call must complete with a corresponding call to the AWAITIO[X|XL] procedure when used with a file that is OPENed nowait. If READUPDATEXL is used, you must call AWAITIOXL to complete the I/O. If READUPDATEX is used, you must call AWAITIOX or AWAITIOXL to complete the I/O. If READUPDATE is used, you may use AWAITIO or AWAITIOX or AWAITIOXL to complete the I/O.

▲ **WARNING.** When using nowait file I/O, data corruption might occur if the READ buffer is modified before the AWAITIOX that completes the call..

- Default locking mode action

  If the default locking mode is in effect when a call to READUPDATE[X|XL] is made to a locked file or record, but the *filenum* of the locked file differs from the *filenum* in the call, the caller of READUPDATE[X|XL] is suspended and queued

in the “locking” queue behind other processes attempting to access the file or record.

Note. A deadlock condition occurs if a call to READUPDATE[X|XL] is made by a process having multiple opens on the same file and the *filenum* used to lock the file differs from the *filenum* supplied to READUPDATE[X|XL].

- Alternate locking mode action

  If the alternate locking mode is in effect when READUPDATE[X|XL] is called and the file is locked but not through the file number supplied in the call, the call is rejected with error 73 (“file is locked”).

- Lock mode by SETMODE

  The locking mode is specified by the SETMODE procedure, function 4.

- Value of the current key and current-key specifier

  For key-sequenced, relative, and entry-sequenced files, random processing implies that a designated record must exist. Therefore, positioning for READUPDATE[X|XL] is always to the record described by the exact value of the current key and current-key specifier. If such a record does not exist, the call to READUPDATE[X|XL] is rejected with a file-system error 11 (“record does not exist”). This is unlike sequential processing through READ[X] where positioning can be by approximate, generic, or exact key value.

Disk File Considerations

- Large data transfers

  For READUPDATEX only, large data transfers (more than 4096 bytes), can be enabled by using SETMODE function 141. See Table 14-4 on page 14-63.

- Record does not exist

  If the position specified for the READUPDATE[X|XL] operation does not exist, the call is rejected with error 11. (The positioning is specified by the exact value of the current key and current-key specifier.)

- Structured files

  - READUPDATE[X|XL] without selecting a specific record

    If the call to READUPDATE[X|XL] immediately follows a call to KEYPOSITION, the call to KEYPOSITION must specify exact positioning mode in the *positioning-mode* parameter and the length of the entire key in the *length-word* parameter.

    If the call to READUPDATE[X|XL] immediately follows a call to KEYPOSITION where a nonunique alternate key is specified, the READUPDATE[X|XL] fails. A subsequent call to FILE_GETINFOL_ or FILE_GETINFO_ or FILEINFO shows that an error 46 (invalid key) occurred. However, if an intermediate call to

READ or READLOCK is made, the call to READUPDATE[X|XL] is permitted because a unique record is identified.

- indicators after READUPDATE[X|XL]

  After a successful READUPDATE[X|XL], the current-state indicators are unchanged (current- and next-record pointers).

- Unstructured disk files
  - unstructured files

    For a READ[X] from an unstructured disk file, data transfer begins at the position indicated by the current-record pointer. A call to READUPDATE[X|XL] typically follows a call to POSITION that sets the current-record pointer to the desired relative byte address.

  - pointer action for unstructured files is unaffected.
  - of *count-read* for unstructured files

    After a successful call to READUPDATE[X|XL] to an unstructured file, the value returned in *count-read* is determined by:

    ```
    count-read := $MIN(read-count,EOF - next-record
                       - next-record pointer)
    ```

  - number of bytes read

    If the unstructured file is created with the odd unstructured attribute (also known as ODDUNSTR) set, the number of bytes read is exactly the number specified with *read-count*. If the odd unstructured attribute is not set when the file is created, the value of *read-count* is rounded up to an even value before the READUPDATE[X|XL] is executed.

    You set the odd unstructured attribute with the FILE_CREATE_ , FILE_CREATELIST_ , or CREATE procedure, or with the File Utility Program (FUP) SET and CREATE commands.

Interprocess Communication Considerations

- Replying to messages

  Each message read in a call to READUPDATE[X|XL], including system messages, must be replied to in a corresponding call to the REPLY[X|XL] procedure.

- Queuing several messages before replying

  Several interprocess messages can be read and queued by the application process before a reply must be made. The maximum number of messages that the application process expects to read before a corresponding reply is made must be specified in the *receive-depth* parameter to the FILE_OPEN_ or OPEN procedure.

- If $RECEIVE is opened with *receive-depth* = 0, only READ[X]s can be performed, and READUPDATE[X|XL] and REPLY[X|XL] fail with error 2 (“operation not allowed on this type of file”).
- Message tags when replying to queued messages

  If more than one message is to be queued by the application process (that is, *receive-depth* > 1), a message tag that is associated with each incoming message must be obtained in a call to the FILE_GETRECEIVEINFOL_ (or FILE_GETRECEIVEINFO_ or LASTRECEIVE or RECEIVEINFO) procedure following each call to READUPDATE[X|XL]. To direct a reply back to the originator of the message, the message tag associated with the incoming message is passed to the system in a parameter to the REPLY[X|XL] procedure. If messages are not to be queued, it is not necessary to call FILE_GETRECEIVEINFOL_ or FILE_GETRECEIVEINFO_.

Considerations for READUPDATEX and READUPDATEXL

- The buffer and count transferred may be in the user stack or in an extended data segment. The buffer and count transferred cannot be in the user code space.
- If the buffer or count transferred is in a selectable extended data segment, the segment must be in use at the time of the call. Flat segments allocated by a process are always accessible to the process.
- The size of the transfer is subject to current restrictions for the type of file.
- If the file is opened for nowait I/O, and the buffer is in an extended data segment, you cannot deallocate or reduce the size of the extended data segment before the I/O completes with a call to AWAITIOX or AWAITIOXL or is canceled by a call to CANCEL or CANCELREQ or CANCELREQL.
- If the file is opened for nowait I/O, you must not modify the buffer before the I/O completes with a call to AWAITIOX or AWAITIOXL. This also applies to other processes that may be sharing the segment. It is the application’s responsibility to ensure this.
- If the file is opened for nowait I/O, and the I/O has been initiated with these routines, the I/O must be completed with a call to AWAITIOX or AWAITIOXL (not AWAITIO).
- If the file is opened for nowait I/O, a selectable extended data segment containing the buffer need not be in use at the time of the call to AWAITIOX or AWAITIOXL.
- Nowait I/O initiated with these routines may be canceled with a call to CANCEL or CANCELREQ or CANCELREQL. The I/O is canceled if the file is closed before the I/O completes or AWAITIOX or AWAITIOXL is called with a positive time limit and specific file number and the request times out.
- A file opened by FILE_OPEN_ uses direct I/O transfers by default except on NSAA systems; you can use SETMODE 72 to force the system to use an intermediate buffer in the process file segment (PFS—also known as system buffers) for I/O

transfers. A file opened by OPEN uses a PFS buffer for I/O transfers, except for large transfers to DP2 disks.

- If the extended address of the buffer is odd, bounds checking rounds the address to the next lower word boundary and checks an extra byte as well. The odd address is used for the transfer.

Errors for READUPDATEX and READUPDATEXL

In addition to the errors currently returned from READUPDATE, error 22 is returned from READUPDATEX and READUPDATEXL when:

- The address of a parameter is extended, but either the extended data segment is invalid or the address is for a selectable segment that is not in use at the time of the call.
- The address of a parameter is extended, but it is an absolute address and the caller is not privileged.

Related Programming Manuals

For programming information about the READUPDATE[X] file-system procedure, see the *Guardian Programmer's Guide* and the *Enscribe Programmer's Guide*.

Note. The READUPDATEXL procedure is supported on systems running J06.07 and later J-series RVUs and H06.18 and later H-series RVUs.

READUPDATELOCK[X] Procedures

Summary

The READUPDATELOCK[X] procedures are used for random processing of records in a disk file. READUPDATELOCK is intended for use with 16-bit addresses, while READUPDATELOCKX is intended for use with 32-bit extended addresses. Therefore, the data buffer for READUPDATELOCKX can be either in the caller's stack segment or any extended data segment.

READUPDATELOCK[X] locks, then reads the record from the current position in the file in the same manner as the combination of LOCKREC and READUPDATE[X]. READUPDATELOCK[X] is intended for reading a record after calling POSITION or KEYPOSITION, possibly in anticipation of a subsequent call to the WRITEUPDATE[X] or WRITEUPDATEUNLOCK[X] procedure.

A call to READUPDATELOCK[X] is functionally equivalent to a call to LOCKREC followed by a call to READUPDATE[X]. However, less system processing is incurred when one call is made to READUPDATELOCK[X] rather than two separate calls to LOCKREC and READUPDATE[X].

Syntax for C Programmers

```
#include <cextdecs(READUPDATELOCK)>

_cc_status READUPDATELOCK ( short filenum
                          ,short _near *buffer
                          ,unsigned short read-count
                          ,[unsigned short _near *count-read ]
                          ,[ __int32_t tag ] );


#include <cextdecs(READUPDATELOCKX)>

_cc_status READUPDATELOCKX ( short filenum
                           ,char _far *buffer
                           ,unsigned short read-count
                           ,[unsigned short _far *count-read ]
                             ,[ __int32_t tag ] );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.
- The function value returned by READUPDATELOCK[X], which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL READUPDATELOCK[X] ( filenum                    ! i
                        ,buffer                     ! o
                        ,read-count                 ! i
                        ,[ count-read ]             ! o
                        ,[ tag ] );                 ! i
```

Parameters

filenum input

INT:value (Use with READUPDATELOCK and READUPDATELOCKX)

is the number of an open file that identifies the file to be read.

buffer output

INT:ref:* (Use with READUPDATELOCK)
STRING .EXT:ref:* (Use with READUPDATELOCKX)

is an array where the information read from the file returns.

read-count input

INT:value (Use with READUPDATELOCK and READUPDATELOCKX)

is the number of bytes to be read {0:4096}.

count-read output

INT:ref:1 (Use with READUPDATELOCK)

INT .EXT:ref:1 (Use with READUPDATELOCKX)

is for wait I/O only. *count-read* returns a count of the number of bytes returned from the file into *buffer*.

tag input

INT(32):value (Use with READUPDATELOCK and READUPDATELOCKX)

is for nowait I/O only. *tag* is a value you define that uniquely identifies the operation associated with this READUPDATELOCK[X].

Note. The system stores the *tag* value until the I/O operation completes. The system then returns the *tag* information to the program in the *tag* parameter of the call to AWAITIO[X], thus indicating that the operation completed. If READUPDATELOCKX is used, you must call AWAITIOX to complete the I/O. If READUPDATELOCK is used, you may use either AWAITIO or AWAITIOX to complete the I/O.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILE_GETINFO_ or FILEINFO).

< (CCL) is also returned following a successful read with an insertion-ordered alternate key path if the alternate key value of the current record is equal to the alternate key value in this record along that path. A call to FILE_GETINFO_ or FILEINFO shows that an error 551 occurred; this error is advisory only and does not indicate an unsuccessful read operation.

= (CCE) indicates that the READUPDATELOCK[X] is successful.

> (CCG) does not return from READUPDATELOCK[X] for disk files.

Considerations

- READUPDATELOCK versus READUPDATELOCKX

  Use READUPDATELOCK when the buffer has a 16-bit address, and use READUPDATELOCKX when the buffer has a 32-bit extended address. Therefore, the data buffer for READUPDATELOCKX can be either in the caller's stack segment or any extended data segment.

- Nowait I/O and READUPDATELOCK[X]

  The READUPDATELOCK[X] procedure must complete with a corresponding call to the AWAITIO[X] procedure when used with a file that is opened nowait. If

READUPDATELOCKX is used, the user must call the AWAITIOX procedure to complete the I/O. If READUPDATELOCK is used, the user may use either AWAITIO or AWAITIOX to complete the I/O.

▲ **WARNING.** When using nowait file I/O, data corruption might occur if the READ buffer is modified before the AWAITIOX that completes the call.

- If READUPDATELOCK[X] is performed on nondisk files, an error is returned.
- Random processing

  For key-sequenced, relative, and entry-sequenced files, random processing implies that a designated record must exist. Therefore, positioning for READUPDATELOCK[X] is always to the record described by the exact value of the current key and current-key specifier. If such a record does not exist, the call to READUPDATELOCK[X] is rejected with file-system error 11.
- See Considerations on page 8-20.
- See Considerations on page 13-27.
- Queue files

  To use READUPDATELOCK[X], a queue file must be opened with write access and with a *sync-or-receive-depth* of 0.

Considerations for READUPDATELOCKX Only

- The buffer and count transferred may be in the user stack or in an extended data segment. The buffer and count transferred cannot be in the user code space.
- If the buffer or count transferred is in a selectable extended data segment, the segment must be in use at the time of the call. Flat segments allocated by a process are always accessible to the process.
- If the file is opened for nowait I/O, and the buffer is in an extended data segment, you cannot deallocate or reduce the size of the extended data segment before the I/O completes with a call to AWAITIOX or is canceled by a call to CANCEL or CANCELREQ.
- If the file is opened for nowait I/O, you must not modify the buffer before the I/O completes with a call to AWAITIOX. This also applies to other processes that may be sharing the segment. It is the application's responsibility to ensure this.
- If the file is opened for nowait I/O, and the I/O has been initiated with these routines, the I/O must be completed with a call to AWAITIOX (not AWAITIO).
- If the file is opened for nowait I/O, a selectable extended data segment containing the buffer need not be in use at the time of the call to AWAITIOX.
- Nowait I/O initiated with these routines may be canceled with a call to CANCEL or CANCELREQ. The I/O is canceled if the file is closed before the I/O completes or

AWAITIOX is called with a positive time limit and specific file number and the request times out.

- A file opened by FILE_OPEN_ uses direct I/O transfers by default; you can use SETMODE 72 to force the system to use an intermediate buffer in the process file segment (PFS) for I/O transfers. A file opened by OPEN uses a PFS buffer for I/O transfers, except for large transfers to DP2 disks.
- If the extended address of the buffer is odd, bounds checking rounds the address to the next lower word boundary and checks an extra byte as well. The odd address is used for the transfer.

Errors for READUPDATELOCKX Only

In addition to the errors returned from READUPDATELOCK, error 22 is returned from READUPDATELOCKX when:

- The address of a parameter is extended, but either the extended data segment is invalid or the address is for a selectable segment that is not in use at the time of the call.
- The address of a parameter is extended, but it is an absolute address and the caller is not privileged.

OSS Considerations

- This procedure operates only on Guardian objects. If an OSS file is specified, error 2 is returned.

Example

```
CALL READUPDATELOCK ( IN^FILE , INBUFFER , 72 , NUM^READ );
```

Related Programming Manuals

For programming information about the READUPDATELOCK file-system procedure, see the *Enscribe Programmer's Guide*.

RECEIVEINFO Procedure (Superseded by FILE_GETRECEIVEINFO[L]_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The RECEIVEINFO procedure is used to obtain the 4-word process ID, message tag, error recovery (sync ID), and request-related (file number, read count, and I/O type) information associated with the last message read from the $RECEIVE file. Because this information is contained in the file's main-memory resident access control block (ACB), the application process is not suspended by a call to RECEIVEINFO.

Note that the first two parameters to RECEIVEINFO (*process-id* and *message-tag*) duplicate the parameters to the LASTRECEIVE procedure.

Note. To ensure that you receive valid information about the last message, call RECEIVEINFO before you perform another READUPDATE on $RECEIVE. If you received an error condition on the last message, call FILEINFO or FILE_GETINFO_ to obtain the error value before you call RECEIVEINFO.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
CALL RECEIVEINFO ( [ process-id ]              ! o
                  ,[ message-tag ]             ! o
                  ,[ sync-id ]                 ! o
                  ,[ filenum ]                 ! o
                  ,[ read-count ]              ! o
                  ,[ iotype ] );               ! o
```

Parameters

process-id output

INT:ref:4

returns the 4-word process ID of the process that sent the last message read through the $RECEIVE file. If the process is of the named form, and thus is in the destination control table (DCT), the information returned consists of:

[0:2]	$*process-name*
[3].<0:3>	Reserved
.<4:7>	Processor number where the process is executing
.<8:15>	PIN assigned by the operating system to identify the process in the processor

If the process is of the unnamed form, and thus is not in the destination control table (DCT), the information returned consists of:

[0:2]	*creation-time-stamp*
[3].<0:3>	Reserved
.<4:7>	Processor number where the process is executing
.<8:15>	PIN assigned by the operating system to identify the process in the processor

process-id (continued)

If the process ID is of the network form, the information returned consists of:

[0] .<0:7>	"\" (ASCII backslash)
[0] .<8:15>	System number
[1:2]	Process name
[3] .<0:3>	Reserved
.<4:7>	Processor number where the process is executing
.<8:15>	PIN assigned by the operating system to identify the process in the processor

message-tag output

INT:ref:1

is used when the application process performs message queuing. *message-tag* returns a value that identifies the request message just read among other requests

currently queued. To associate a reply with a given request, *message-tag* is passed in a parameter to the REPLY procedure. The returned value of *message-tag* is an integer between zero and *receive depth* -1, inclusive, that is not currently used as a message tag. When a reply is made, its associated message tag value is made available for use as a message tag for a subsequent request message.

sync-id output

INT(32):ref:1

returns the sync ID associated with this message. If the received message is a system message, this parameter is valid only if the message is associated with a specific file open; otherwise this parameter is not applicable and should be ignored.

filenum output

INT:ref:1

returns the file number of the file in the requesting process associated with this message. If the received message is a system message that is not associated with a specific file open, this parameter contains -1.

read-count output

INT:ref:1

returns the number of bytes requested in reply to the message. If the message is the result of a request made in a call to WRITE[X], *read-count* will be 0. If the message is the result of a request made in a call to WRITEREAD[X], *read-count* is the same as the read count value passed by the requester to WRITEREAD[X].

iotype output

INT:ref:1

returns a value indicating the data operation last performed by the message sender:

0 Not a data message (system message)
1 Sender called WRITE
2 Sender called READ
3 Sender called WRITEREAD

Condition Code Settings

< (CCL) indicates that $RECEIVE is not open.

= (CCE) indicates that RECEIVEINFO is successful.

> (CCG) does not return from RECEIVEINFO.

Considerations

- Process ID and RECEIVEINFO

  The 4-word process ID returned by RECEIVEINFO following receipt of a process open, close, CONTROL, SETMODE, SETPARAM, RESETSYNC, or CONTROLBUF system message, or a data message, identifies the process associated with the operation.

- When the high-order three words of process ID are zero

  The high-order three words of the process ID are zero following the receipt of system messages other than process open, close, CONTROL, SETMODE, RESETSYNC, and CONTROLBUF.

- Synthetic process ID

  If HIGHREQUESTERS is enabled for the calling process (either because the ?HIGHREQUESTERS flag is set in the program file or because the caller used FILE_OPEN_ to open $RECEIVE) and the last message was sent by a high-PIN process, then the returned process ID is as described above except that the value of the PIN is 255. This form of the process ID is referred to as a *synthetic* process ID. It is not a full identification of the process but it is normally sufficient for distinguishing, for example, one requester from another requester. For further details, see the *Guardian Programmer's Guide*.

- Remote opener with a long process file name

  If the calling process used FILE_OPEN_ to open $RECEIVE and did not request to receive C-series format messages, and if the last message read from $RECEIVE is from a remote process that has a process name consisting of more than five characters, then the value of *process-id* returned by RECEIVEINFO is undefined.

- Sync ID definition

  A sync ID is a doubleword, unsigned integer. Each process file that is open has its own sync ID. Sync IDs are not part of the message data; rather, the sync ID value associated with a particular message is obtained by the receiver of a message by calling the RECEIVEINFO procedure. A file's sync ID is set to zero at file open and when the RESETSYNC procedure is called for that file (RESETSYNC can be called directly or indirectly through the CHECKMONITOR procedure). For information about checkpointing, see the *Guardian Programmer's Guide*.

  When a request is sent to a process (that is, CONTROL, CONTROLBUF, close, open, SETMODE, WRITE, or WRITEREAD to a process file), the requestor's sync ID is incremented by 1 just before to the request is sent. (Therefore, a process's first sync ID subsequent to an open has a value of 0.)

- Duplicate requests

  The *sync-id* parameter allows the server process (that is, the process reading $RECEIVE) to detect duplicate requests from requester processes. Such duplicate requests are caused by a backup requester process reexecuting the latest request of a failed primary requester process.

Note. Neither a CANCELREQ or AWAITIO timeout completion have any affect on the sync ID (that is, it will be an ever-increasing value).

Also, the sync ID is independent of the `sync-depth` parameter to OPEN.

- Server process identifying separate opens by the same requester

  The *filenum* parameter allows the server process to identify separate opens by the same requester process. The value returned in *filenum* is the same as the file number used by the requester to make this request.

Example

```
CALL RECEIVEINFO (PROCID , , , REQ^FNUM , REQ^READCOUNT);
```

Related Programming Manual

For programming information about the RECEIVEINFO file-system procedure, see the *Guardian Programmer's Guide*.

REFPARAM_BOUNDSCHECK_ Procedure

Summary

Note. This procedure is declared only in the EXTDECS0 file.

The REFPARAM_BOUNDSCHECK_ procedure checks the validity of parameter addresses passed to the procedure that calls it. Bounds checking performed by the system is enough for most applications. This procedure, however, provides additional checks for those few applications that might need it.

Primarily, REFPARAM_BOUNDSCHECK_ verifies that a specified memory area is valid for a specified type of access (read only or read/write). Optionally, it also verifies

that the specified memory area does not overlap the part of the process stack occupied by the calling procedure and any of the procedures it calls.

Syntax for C Programmers

```
#include <cextdecs(REFPARAM_BOUNDSCHECK_)>

short REFPARAM_BOUNDSCHECK_ ( void _far *start-address
                             ,__int32_t area-len
                             ,void _far *framestart
                             ,short flags )
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

To exclude the stack area from the area that REFPARAM_BOUNDSCHECK_ accepts as valid in a C or C++ program, you pass the `_frame_edge(first,last)` function as the *frame-start* parameter to REFPARAM_BOUNDSCHECK_. The *first* and *last* parameters are the names of the first and last parameters to the calling function. For example:

```
#include <cextdecs(REFPARAM_BOUNDSCHECK_)>

short tstbnd (short *alpha, long long omega)
{
  if ( REFPARAM_BOUNDSCHECK_( alpha,2,
                              _frame_edge(alpha,omega),0))
    return -1;
  return 0;
}
```

A function with no parameters cannot use `_frame_edge()`, nor can a C++ function whose first parameter is a reference (that is, not a value or a pointer).

Syntax for TAL Programmers

```
error := REFPARAM_BOUNDSCHECK_ ( start-address          ! i
                                ,area-length            ! i
                                ,framestart             ! i
                                ,flags );               ! i
```

Parameters

error returned value

INT

returns one of these values:

0 No error. The procedure successfully executed; the specified memory area is in bounds. For a discussion of what it means to be in bounds, see “Considerations”.

1 The specified memory area is out of bounds. Accessing the area might cause an addressing trap or system-generated nondeferrable signal.

2 The address is in a read-only area and the check was made for read/write access. The effect of attempting to write to the area depends on whether your process is a native process or a TNS process: for a native process, the system might deliver a nondeferrable signal to the process; for a TNS process, the write operation might not take effect.

3 The address area is in bounds in an extensible segment, but disk space for the extensible segment could not be allocated. If you try to write to this area, the effect depends on whether your process is a TNS process or a native process: a TNS process might terminate with a “no memory available” trap (trap 12); a native process might receive a SIGNOMEM signal.

4 The *start-address* parameter points to a location on the TOSSTACK but the calling procedure is not executing on the TOSSTACK. This error can occur only on a TNS processor.

5 An absolute address was supplied in *start-address* for a native process but *flags*.<14> was not set to allow an absolute address.

start-address input

STRING .EXT:ref:*

identifies the start of the data area to be bounds checked.

When REFPARAM_BOUNDSCHECK_ is called from a native process, *start-address* cannot be a relative segment 1 or 2 address. Neither can it be an absolute address unless *flags*.<14> is set to allow this.

When REFPARAM_BOUNDSCHECK_ is called from a TNS process, *start-address* can be an absolute address that points to the TNS stack.

area-length input

INT(32):value

is the length of the area to be checked. It is a 31-bit value. On a native processor, the area must not span separately allocated logical areas even if the areas are contiguous in virtual memory.

If *area-length* is zero (0D), *start-address* is not checked and REFPARAM_BOUNDSCHECK_ returns without error.

framestart input

EXTADDR:value

excludes the stack area of the calling procedure from the area that REFPARAM_BOUNDSCHECK_ accepts as valid.

If *framestart* is zero, REFPARAM_BOUNDSCHECK_ verifies only that the specified memory area is valid. It makes no attempt to verify that the specified memory area does not overlap the part of the stack used by the calling process.

If *framestart* is nonzero, it specifies one edge of a region to be excluded from the valid area. For a TNS process, the excluded area begins at the address in *framestart* and extends 600 bytes past the L-register setting of the procedure that calls REFPARAM_BOUNDSCHECK_. For a native process, the excluded area begins at the start of the stack area and extends to (but does not include) the address specified by *framestart*. Stack-frame overlap detection is possible only for procedures running on the default stack of the process; it is not performed if *framestart* designates an address on a user thread stack.

In pTAL, the correct value to pass in *framestart* is obtained using the $PARAMSTART function. In TAL, you use the $XADR function. A set of DEFINEs are available to mask the differences between pTAL and TAL. See "Considerations" for details.

flags input

INT:value

specifies options. The bits indicate:

<0:13> must be zero.

<14> = 0 specifies that absolute addressing is not allowed for native processes.

= 1 allows absolute addressing even if REFPARAM_BOUNDSCHECK_ is called from a native process. No checking is performed and REFPARAM_BOUNDSCHECK_ returns without error.

<15> = 0 checks the area for read/write access.

= 1 checks the area for read-only access.

Considerations

- To pass the tests performed by REFPARAM_BOUNDSCHECK_, a correct reference parameter must meet this criteria:
  - The reference address must start within a range that is mapped into the user's logical address space.
  - The entire referenced area specified by *start-address* and *area-length* is within the same logical addressing space as the *start-address*. That is, it must be entirely within one of these: the TNS data stack, the main stack, or a single selectable extended data segment. In the case of TNS code segments, the address must be within a single code space if the TNS code segment spans multiple code spaces.
  - The reference parameter must be within an unprotected logical addressing space. For TNS processors, this addressing space is either the TNS data

stack, an unprivileged selectable extended data segment, or certain relative code segments. On native processors, this area can be the TNS data stack, the main stack, an unprivileged selectable extended data segment, or any of the code areas mapped into KUSEG.

- For CALLABLE procedures running on the TNS data stack, no part of the reference parameter can be within an area that starts at the base of the CALLABLE procedure's stack and extends to 0 locations beyond the L register of the CALLABLE procedure. This is true only for procedures that run on a TNS processor and are not running on the TOSSSTACK, or for TAL procedures running on a native processor.
- For CALLABLE procedures with parameters within a valid code area, the parameter address is valid only if the CALLABLE procedure is making a read-only reference.

- If the address area is in bounds in an extensible segment and disk space for the extensible segment needs to be allocated, REFPARAM_BOUNDSCHECK_ allocates more space for the segment. If no space is available, REFPARAM_BOUNDSCHECK_ returns error 3.
- To exclude address references from the stack area used by the calling procedure, use DEFINES with the *framestart* parameter as follows:
  1. Invoke the _FRAME_EDGE_DEF DEFINE among the declarations in the procedure.
  2. Pass the names of the first and last parameters of the procedure to the _FRAME_EDGE_DEF DEFINE. If the procedure has no parameters, invoke _FRAME_EDGE_DEF ('L' - 2).
  3. Pass _FRAME_EDGE as the *framestart* parameter to REFPARAM_BOUNDSCHECK_.
- In a multithreaded process that uses multiple stack areas, checking is performed only for the thread that calls REFPARAM_BOUNDSCHECK_.

Example

```
INT PROC TSTBND ( alpha,omega);
  INT .EXT alpha;
  FIXED omega;
BEGIN
  _FRAME_EDGE_DEF ( alpha,omega );
  IF REFPARAM_BOUNDSCHECK_(alpha,2D,_FRAME_EDGE,0) THEN
    RETURN -1;
  RETURN 0;
END;
```

REFRESH Procedure (Superseded by DISK_REFRESH_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development. On G-series RVUs, the function provided by both the REFRESH and DISK_REFRESH_ procedures is no longer needed.

The REFRESH procedure is used to write control information to the associated disk volume. REFRESH always writes out the control information contained in file control blocks (FCBs), such as end-of-file (EOF) pointers. Only the data and control information that is not already on disk is written.

REFRESH also writes all dirty (that is, modified) cache blocks to disk. The writing of cache blocks takes priority over all other disk activity and can severely impact response time on the disk volume. For this reason, the REFRESH procedure should not be used when performance of other programs is critical.

On RVUs preceding G00,The REFRESH procedure can be used when a volume is brought down (for example, immediately before a system load or PUP DOWN ! command) but should not be used at other times. On these RVUs,The REFRESH procedure or the equivalent Peripheral Utility Program (PUP) REFRESH command should be performed on all volumes before a total system shutdown. On G-series RVUs, the REFRESH procedure is not needed because the system performs the equivalent operation automatically for each disk volume when it is brought down and at system shutdown.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
error := REFRESH ( [volname] );            ! i
```

Parameters

error returned value

INT

returns a file-system error number indicating the outcome of the call. (For a list of all file-system errors, see the *Guardian Procedure Errors and Messages Manual*.)

volname input

INT:ref:12

specifies a volume whose associated FCBs should be written to disk. $*volname* must be specified as a full 12-word internal form *file-name*, blank filled. *volname* can be either:

$*volname*

or

sysnumvolname

If omitted, all FCBs for all volumes are written to their respective disks.

Considerations

- When REFRESH is called without a *volname*, the error return is always 0.
- Because calling the REFRESH procedure can severely impact response time on the specified disk volume, these actions might be considered as alternatives:
  - When creating a file using FILE_CREATE_ , FILE_CREATELIST_ , or CREATE, select the option that causes the file label to be written immediately to disk whenever the EOF value changes.
  - Use SETMODE function 95 to cause the dirty cache buffers of a specified file to be written to disk.

Example

```
ERROR := REFRESH;  ! refresh FCBs for all volumes.
```

REMOTEPROCESSORSTATUS Procedure

Summary

The REMOTEPROCESSORSTATUS procedure supplies the status of processor modules in a particular system in a network.

Syntax for C Programmers

```
#include <cextdecs(REMOTEPROCESSORSTATUS)>

__int32_t REMOTEPROCESSORSTATUS ( short sysnum );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
status := REMOTEPROCESSORSTATUS ( sysnum );          ! i
```

Parameters

status returned value

INT(32)

returns two words indicating the processor status.

The most significant word (MSW) is the highest processor number in the remote system plus one. The least significant word (LSW) is a bit mask for processor availability. If LSW is all zeros, the number of processors is not available and you should ignore any value in MSW.

Word [0]	MSW 0D if the remote system is nonexistent or unavailable
Word [1]	LSW, bit mask: 1 the processor is up 0 the processor is down or nonexistent

sysnum input

INT:value

is the number of a particular system in a network whose processor modules' status is returned.

Considerations

- Where to find the system number

  The system number for a particular system whose name is known can be obtained from the LOCATESYSTEM procedure.

- Equivalencing the two words of *status*

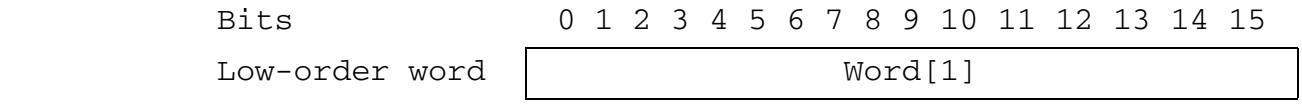
  The two words of *status* can be separated by the usual technique of equivalencing INT variables to the high- and low-order words. For example, a Tandem Application Language procedure that calls REMOTEPROCESSORSTATUS might contain these declarations:

```
INT(32) STATUS;
INT NUM^PROCESSORS = STATUS;         ! high-order word.
INT BIT^MASK = NUM^PROCESSORS + 1;   ! low-order word.
```

  For an explanation of equivalenced variables, see the *TAL Reference Manual*.

- Low-order word of *status*

  The bits in the low-order word are ordered from 0 to 15, from left to right (the processor number corresponds to the bit number):



- Using *status* for local processors

  REMOTEPROCESSORSTATUS can also be used to obtain the status of local processors:

```
INT(32) MY^PROCESSOR^STATUS;
MY^PROCESSOR^STATUS := REMOTEPROCESSORSTATUS
                       ( MYSYSTEMNUMBER );
```

- Caution using *status* for network process recovery

  Using REMOTEPROCESSORSTATUS for network process recovery can lead to peculiar timing problems. This procedure relies on Expand's NCP table for processor status. This table is not updated immediately with current processor status, so there is a window of time during which a check of REMOTEPROCESSORSTATUS indicates an erroneous processor UP condition. A subsequent call to PROCESS_CREATE_ goes directly to the downed processor and fails. You must take care to avoid this possible window problem when writing code to handle retries and other related problems.

REMOTETOSVERSION Procedure

Summary

The REMOTETOSVERSION procedure identifies which version of the operating system is running on a remote system.

Syntax for C Programmers

```
#include <cextdecs(REMOTETOSVERSION)>

short REMOTETOSVERSION ( [ short sysid ] );
```

Syntax for TAL Programmers

```
tos-version := REMOTETOSVERSION [ ( sysid ) ];       ! i
```

Parameters

tos-version returned value

INT

returns a value of the form:

<0:7> an uppercase ASCII letter indicating the version of the operating system
<8:15> a binary number specifying the release number of the version

ASCII Letter	Operating-System Version
N	Dnn
P	Fnn
Q	Gnn
R	Hnn

Zero is returned if the system *sysid* does not exist or is inaccessible.

sysid input

INT:value

is the system number that identifies the system for which the operating system version is returned. If *sysid* is omitted, it defaults to the local system.

Example

```
REMOTE^VERSION := REMOTETOSVERSION ( SYSTEM^NUM );
```

For example, if the operating-system version is D10, the returned value contains “N” in bits <0:7> and binary 10 in bits <8:15>.

Related Programming Manual

For programming information about the REMOTETOSVERSION procedure, see the *Guardian Programmer’s Guide*.

RENAME Procedure (Superseded by FILE_RENAME_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The RENAME procedure is used to change the name of a disk file that is open. If the file is temporary, assigning a name causes the file to be made permanent.

A call to the RENAME procedure is rejected with an error indication if there are incomplete nowait operations pending on the specified file.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
CALL RENAME ( filenum                  ! i
            ,new-name );               ! i
```

Parameters

filenum input

INT:value

is the number of an open file that identifies the file to be renamed.

new-name input

INT:ref:12

is an array containing the internal-format file name to be assigned to the disk file, as follows:

[0:3]	$*volname* (blank-fill)
	or
	sysnum volname (blank-fill)
[4:7]	*subvolname* (blank-fill)
[8:11]	*file-id* (blank-fill)

Condition Code Settings

< (CCL) indicates that an error occurred (call FILEINFO or FILE_GETINFO_).

= (CCE) indicates that the rename operation is successful.

> (CCG) indicates that the file is not a disk file.

Considerations

- Purge access for RENAME

  The caller must have purge access to the file for the RENAME to be successful. Otherwise, the RENAME is rejected with file-system error 48, “security violation.”

- Volume specification for *new-name*

  The volume specified in *new-name* must be the same as the volume specified when opening the file. Neither the volume name nor the system name can be changed by RENAME.

- *sysnum* specification for *new-name*

If *sysnum* is specified as part of *new-name*, it must be the same as the system number used when the file was initially opened.

- Partitioned files

  When the primary partition of a partitioned file is renamed, the file system automatically renames all other partitions located anywhere in the network.

- Renaming a file audited by the Transaction Management Facility (TMF)

  The file to be renamed cannot be a file audited by TMF. An attempt to rename such a file fails with file-system error 80 (invalid operation attempted on audited file or nonaudited disk volume).

- Structured files with alternate keys

  If the primary-key file is renamed, it is linked with the alternate-key file. If you rename the alternate-key file and then try to access the primary-key file, file-system error 4 occurs, because the primary-key file is still linked with the old name for the alternate-key file. You can use the FUP ALTER command to correct this problem.

Safeguard Considerations

For information on files protected by Safeguard, see the *Safeguard Reference Manual*.

OSS Considerations

- An OSS file cannot be renamed. Error 564 (operation not supported on this file type) occurs when an attempt is made to rename an OSS file.

REPLY[X|XL] Procedures

Summary

The REPLY[X|XL] procedures are used to send a reply message to a message received earlier in a corresponding call to READUPDATE[X|XL] on the $RECEIVE file.

The REPLY[X|XL] procedures can be called even if there are incomplete nowait I/O operations pending on $RECEIVE.

Syntax for C Programmers

```
#include <cextdecs(REPLY)>

_cc_status REPLY ( [ short _near *buffer ]
                  ,[ unsigned short write-count ]
                  ,[ unsigned short _near *count-written ]
                  ,[ short message-tag ]
                  ,[ short error-return ] );


#include <cextdecs(REPLYX)>

_cc_status REPLYX ( [ const char _far *buffer ]
                   ,[ unsigned short write-count ]
                   ,[ unsigned short _far *count-written ]
                   ,[ short message-tag ]
                   ,[ short error-return ] );


#include <cextdecs(REPLYXL)>

short REPLYXL ( [ const char _far *buffer ]
                ,[ __int32_t  write-count ]
                ,[ __int32_t _far *count-written ]
                ,[ short message-tag ]
                ,[ short error-return ] );
```

- The function value returned by REPLY[X], which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL REPLY[X] ( [ buffer ]                          ! i
              ,[ write-count ]                      ! i
              ,[ count-written ]                    ! o
              ,[ message-tag ]                      ! i
              ,[ error-return ] );                  ! i


error:= REPLYXL ( [ buffer ]                        ! i
              ,[ write-count ]                      ! i
              ,[ count-written ]                    ! o
              ,[ message-tag ]                      ! i
              ,[ error-return ] );                  ! i
```

Parameters

error returned value

INT (Use with REPLYXL)

is a file-system error number indicating the outcome of the operation.

0 (FEOK)

indicates a successful operation.

buffer input

INT:ref:* (Use with REPLY)
STRING .EXT:ref:* (Use with REPLYX and REPLYXL)

is an array containing the reply message.

write-count input

INT:value (Use with REPLY and REPLYX)

is the number of bytes to be written ({0:57344}). If omitted, no data is transferred.

INT(32):value (Use with REPLYXL)

is the number of bytes to be written ({0:2097152}). If omitted or 0, no data is returned.

count-written output

INT:ref:1 (Use with REPLY)
INT .EXT:ref:1 (Use with REPLYX)
INT(32):ref:1 (Use with REPLYXL)

returns a count of the number of bytes written to the file.

message-tag input

INT:value (Use with REPLY, REPLYX, and REPLYXL)

is the *message-tag* returned from FILE_GETRECEIVEINFOL_ (or FILE_GETRECEIVEINFO_ or LASTRECEIVE or RECEIVEINFO) that associates this reply with a message previously received. This parameter can be omitted if message queuing is not performed by the application process (that is, FILE_OPEN_ or OPEN procedure *receive-depth* = 1).

error-return input

INT:value (Use with REPLY, REPLYX, and REPLYXL)

is an error indication that is returned, when the originator's I/O operation completes, to the originator associated with this reply. This indication appears to the originator as a normal file-system error return. The originator's condition code is set according to the relative value of *error-return*.

File-system Error	Condition Code Setting
10-32767	CCL (error)
0	CCE (no error)
1-9	CCG (warning)

(Error numbers 300-511 are reserved for user applications; errors numbers 10-255 and 512-32767 are HP errors.)

The *error-return* value is returned in the *last-error* parameter of FILE_GETINFO[L]_ , or in the *error* parameter of FILEINFO, when the originator calls one of these procedures for the associated completion.

If *error-return* is omitted, a value of 0 (no error) is returned to the message originator.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILE_GETINFO_ or FILEINFO).

= (CCE) indicates the REPLY[X] is successful.

> (CCG) does not return from REPLY[X].

Considerations

- Replying to queued messages

  Several interprocess messages can be read and queued by the application process before a reply must be made. The maximum number of messages that the application process expects to read before a corresponding reply must be specified in the *receive-depth* parameter to the FILE_OPEN_ or OPEN procedure.

If $RECEIVE is opened with *receive-depth* = 0, only READ[X] can be called; READUPDATE[X|XL] and REPLY[X] fail with error 2 (“operation not allowed on this type of file”).

- Using the *message-tag*

  If more than one message is queued by the application process (that is, *receive-depth* > 1), a message tag associated with each incoming message must be obtained in a call to the FILE_GETRECEIVEINFOL_ (or FILE_GETRECEIVEINFO_ or LASTRECEIVE or RECEIVEINFO) procedure immediately following each call to READUPDATE[X|XL]. To direct a reply back to the originator of the message, the message tag associated with the incoming message returns to the system in the *message-tag* parameter to the REPLY[X|XL] procedure. If messages are not queued (that is, *sync-or-receive-depth* = 1), the message tag is not needed.

- Error handling

  The *error-return* parameter can be used to return an error indication to the requester in response to the open, CONTROL, SETMODE, and CONTROLBUF, system messages. The error returns to the requester when the associated I/O procedure finishes.

Considerations for REPLYX and REPLYXL

- The buffer and count written may be in the user stack segment or in an extended data segment. They cannot be in the user’s code space.
- If the buffer or count transferred is in a selectable extended data segment, the segment must be in use at the time of the call. Flat segments allocated by a process are always accessible to the process.
- The transfer size is the same as for procedure REPLY.
- If the extended address of the buffer is odd, bounds checking rounds the address to the next lower word boundary and checks an extra byte. The odd address is used for the transfer.

Errors for REPLYX and REPLYXL

In addition to the errors currently returned from REPLY, error 22 is returned from REPLYX and REPLYXL in these cases:

- The address of a parameter is extended, but either the extended data segment is invalid or the address is for a selectable segment that is not in use at the time of the call.
- The address of a parameter is extended, but it is an absolute address and the caller is not privileged.

Example

```
CALL REPLY ( OUT^BUFFER , 512 );
```

Related Programming Manual

For programming information about the REPLY[X] file-system procedure, see the *Guardian Programmer's Guide*.

Note. The REPLYXL procedure is supported on systems running J06.07 and later J-series RVUs and H06.18 and later H-series RVUs.

REPOSITION Procedure (Superseded by FILE_RESTOREPOSITION_ Procedure)

Summary

The REPOSITION procedure is used to position a disk file to a saved position (the positioning information having been saved by a call to the SAVEPOSITION procedure). The REPOSITION procedure passes the positioning block obtained by SAVEPOSITION back to the file system. Following a call to REPOSITION, the disk file is positioned to the point where it was when SAVEPOSITION was called.

A call to the REPOSITION procedure is rejected with an error if any incomplete nowait operations are pending on the specified file.

Syntax for C Programmers

```
#include <cextdecs(REPOSITION)>

_cc_status REPOSITION ( short filenum
                      ,short _near *positioning-block );
```

- The function value returned by REPOSITION, which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL REPOSITION ( filenum                          ! i
                 ,positioning-block );             ! i
```

Parameters

filenum input

INT:value

is the number of an open file that identifies the file to be positioned to a saved position.

positioning-block input

INT:ref:*

indicates a saved position to be repositioned to. This should not be altered by the user.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILE_GETINFO_ or FILEINFO).

= (CCE) indicates that REPOSITION is successful.

> (CCG) indicates that the file is not a disk file.

Considerations

- The REPOSITION procedure cannot be used with Enscribe format 2 files or OSS files larger than approximately 2 gigabytes. If an attempt is made to use the REPOSITION procedure with these files, error 581 is returned. For information on how to perform the equivalent task with Enscribe format 2 files or OSS files larger than approximately 2 gigabytes, see the FILE_RESTOREPOSITION_ Procedure.

RESETSYNC Procedure

Summary

The RESETSYNC procedure is used by the backup process of a process pair after a failure of the primary process. With this procedure, a different series of operations might be performed from those performed by the primary before its failure.

The RESETSYNC procedure clears a process pair's file synchronization block so that the operations performed by the backup are not erroneously related to the operations just completed by the primary process. It is typically used for open files whose file synchronization blocks are not checkpointed after the most recent stack checkpoint.

Note. Typically, RESETSYNC is not called directly by application programs. Instead, it is called indirectly by CHECKMONITOR.

Syntax for C Programmers

```
#include <cextdecs(RESETSYNC)>

_cc_status RESETSYNC ( short filenum );
```

- The function value returned by RESETSYNC, which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL RESETSYNC ( filenum );          ! i
```

Parameters

filenum input

INT:value

is the number of an open file whose synchronization block is to be cleared.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILE_GETINFO_ or FILEINFO).

= (CCE) indicates that RESETSYNC is successful.

> (CCG) indicates that the file is not a disk file.

Considerations

- File number has not been opened

If the RESETSYNC file number does not match the file number of the open file that you are trying to access, the call to RESETSYNC is rejected with file-system error 16.

- Not receiving messages

  If *filenum* designates a process, and if the $RECEIVE file of that process is not opened with receipt of RESETSYNC messages enabled, then the RESETSYNC procedure fails with file-system error 7.

Example

```
CALL RESETSYNC( FILE^NUMBER );
```

RESIZEPOOL Procedure (Superseded by POOL_* Procedures)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development. *POOL procedures are replaced by POOL_* procedures. There is no one-for-one replacement.

RESIZEPOOL changes the size of a pool that was initialized by the DEFINEPOOL procedure.

Syntax for C Programmers

```
#include <cextdecs(RESIZEPOOL)>

short RESIZEPOOL ( short *pool-head
                  ,__int32_t new-pool-size );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
error  :=  RESIZEPOOL ( pool-head                ! i,o
                       ,new-pool-size );         ! i
```

Parameters

error returned value

INT

returns a file-system error code indicating the outcome of the call:

0 Successful call; the size of the specified pool had been changed to *new-pool-size*.

12 The call would shrink the pool too much, leaving less area than that reserved by GETPOOL; the reserved blocks must be returned by a PUTPOOL.

21 An invalid *new-pool-size* was specified.

22 One of the parameters specifies an address that is out of bounds.

29 A required parameter was not supplied.

59 The pool is invalid and cannot be resized.

pool-head input, output

INT .EXT:ref:19

is a 19-word pool header previously initialized through a call to the DEFINEPOOL procedure. RESIZEPOOL updates this header to reflect the new pool size.

new-pool-size input

INT(32):value

is the new size for the pool, in bytes. This number must be a multiple of 4 bytes and cannot be less than 32 bytes or greater than 127.5 megabytes (133,693,440 bytes). The address of the end of the pool is equal to the address of the beginning of the pool plus *new-pool-size*. Pool space overhead and adjustments for alignment do not cause the pool to extend past this boundary.

Considerations

See Considerations on page 4-26.

Related Programming Manual

For programming information about the RESIZEPOOL procedure, see the *Guardian Programmer's Guide*.

RESIZESEGMENT Procedure

Summary

RESIZESEGMENT alters the size of an existing extended data segment (for example, a segment created by SEGMENT_ALLOCATE_).

Syntax for C Programmers

```
#include <cextdecs(RESIZESEGMENT)>

short RESIZESEGMENT ( short segment-id
                     ,__int32_t new-segment-size );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
error := RESIZESEGMENT ( segment-id                    ! i
                        ,new-segment-size );           ! i
```

Parameters

error returned value

INT

returns a file-system error code indicating the outcome of the call, one of:

-2 Unable to allocate page table space (not returned for unaliased segments).

-1 Unable to allocate segment space.

0 Successful call; the size of the specified extended data segment has been changed to *new-segment-size*.

2 *seg-id* specified a nonexistent extended data segment, or the segment is of a type that cannot be resized (see “Considerations”).

12 Indicates one of these conditions.

- Because the extended data segment is a shared segment, the segment cannot be reduced (see “Considerations”).
- Because an I/O to the segment is in progress, the segment cannot be reduced.
- The segment is being resized.
- There is a lockmemory request on the segment.

21 An invalid *new-segment-size* was specified (see *new-segment-size*, below).

24 *seg-id* specified a privileged segment ID (greater than 2047) and the caller was not privileged.

29 A required parameter was not supplied.

43 Disk space could not be allocated to accommodate the *new-segment-size* specified.

45 Either the existing permanent swap file or temporary swap file for the extended data segment is not large enough for the requested *new-segment-size* or the Kernel-Managed Swap Facility (KMSF) has insufficient resources in the processor. This error was caused by a wrong calculation of the primary and secondary extent sizes. For more information on KMSF, see the *Kernel-Managed Swap Facility (KMSF) Manual*.

segment-id input

INT:value

is the segment ID of the extended data segment to be resized (for example, as specified in a call to SEGMENT_ALLOCATE_).

new-segment-size input

INT(32):value

is the new size for the extended data segment, in bytes.

For a flat segment, the value must be in the range 1 byte through the maximum size defined by the *max-size* parameter of the SEGMENT_ALLOCATE_ procedure.

For a selectable segment, the value must be in the range 1 byte through the maximum size defined by the *max-size* parameter of the SEGMENT_ALLOCATE_ procedure. For a selectable segment allocated by the ALLOCATESEGMENT procedure, the maximum *new-segment-size* is 127.5 megabytes (133,693,440 bytes).

Considerations

- These extended data segment types cannot be resized (an attempt to do so results in an error 2):
  - An extended data segment allocated in the I/O space (IOS) (a segment with a segment ID greater than or equal to 3072).
  - A read-only extended data segment (created by calling SEGMENT_ALLOCATE_ with a *segment-type* parameter of %060000 or by calling ALLOCATESEGMENT with a *pin-and-flags* parameter of %060000).

  These types of public absolute references are adjusted by RESIZESEGMENT if required:
  - The message buffer pointers in message system control blocks (except for resident cache segments)
  - The lock addresses in LOCKWAIT elements
  - The dump, comparison, and memory access breakpoint (MAB) addresses in breakpoint (BPT) entries
  - Memory access breakpoints (MABs) set in the deallocated portion of the extended data segment are removed.
- A call to RESIZESEGMENT that shrinks an extended data segment causes the area beyond the *new-segment-size* to be deallocated. Dirty pages are not written back to a permanent swap file.
- A call to RESIZESEGMENT causes disk extents to be allocated or deallocated (for file-backed segments), or page reservations to be increased or decreased (for KMSF-backed segments) in these ways:

Type of segment	At segment allocate	At memory access	At segment resize
File-backed non-extensible segment	File extents are completely allocated.	Number of allocated extents does not change.	Growing: additional extents are allocated. Shrinking: old extents are deallocated (subject to change).

File-backed extensible segment	One file extent is allocated (subject to change).	Additional extents are allocated.	Growing: no additional extents are allocated. Shrinking: old extents are deallocated (subject to change).
KMSF-backed non-extensible segment	All pages are reserved at the maximum size.	Number of allocated pages does not change.	Growing: the number of pages reserved is adjusted to the new size. Shrinking: the number of pages reserved is reduced (subject to change).
KMSF-backed extensible segment	One page is reserved at the maximum size (subject to change).	Additional pages are reserved.	Growing: no additional pages are reserved. Shrinking: the number of pages reserved is reduced if the new size is less than the highest address of accessed memory (subject to change).

- Because segment resizing is an extremely resource intensive operation, users should design their applications so that RESIZESEGMENT is not frequently called. A good rule of thumb is to call RESIZESEGMENT only when changing the size of an extended data segment by more than 128 KB. Changes that resize an extended data segment by less that 20% should also be avoided.
- A shared extended data segment may be resized to a larger size. RESIZESEGMENT does not permit a currently shared extended data segment to be made smaller.

Considerations for Privileged Callers

- Following a call to RESIZESEGMENT, any underlying absolute segments allocated to the specified extended data segment might change if the resize causes the segment to be extended. Privileged users must not use absolute addresses to reference locations in any extended data segment that could be resized.
- Resident cache segments (segment IDs in the range of 2817 through 3071) are not checked for message system buffers. Resident cache segments should only be allocated by the disk process.

Example

```
INT  ERROR;

ERROR := SEGMENT_ALLOCATE_ ( 0, 2048D ):  ! 1 page extended
                                          ! segment
  .
  .
  .
! extend segment to 65 pages
IF ( ERROR := RESIZESEGMENT( 0, 65D * 2048D ) ) THEN...
   ! an error occurred, ERROR has the error code
```

Related Programming Manual

For programming information about the RESIZESEGMENT procedure, see the *Guardian Programmer's Guide*.

14 Guardian Procedure Calls (S)

This section contains detailed reference information for all user-accessible Guardian procedure calls beginning with the letter S. Table 14-1 lists all the procedures in this section.

Table 14-1. Procedures Beginning With the Letter S (page 1 of 2)

Table 14-1. Procedures Beginning With the Letter S (page 2 of 2)

SIGJMP_MASKSET_ Procedure
SIGLONGJMP_ Procedure
SIGNAL_ Procedure
SIGNALPROCESSTIMEOUT Procedure
SIGNALTIMEOUT Procedure
SIGPENDING_ Procedure
SIGPROCMASK_ Procedure
SIGSETJMP_ Procedure
SIGSUSPEND_ Procedure
SSIDTOTEXT Procedure
STACK_ALLOCATE_ Procedure
STACK_DEALLOCATE_ Procedure
STEPMOM Procedure (Superseded by PROCESS_SETINFO_ Procedure)
STOP Procedure (Superseded by PROCESS_STOP_ Procedure)
STRING_UPSHIFT_ Procedure
SUSPENDPROCESS Procedure (Superseded by PROCESS_SUSPEND_ Procedure)
SYSTEMENTRYPOINT_RISC_ Procedure
SYSTEMENTRYPOINTLABEL Procedure

SAVEPOSITION Procedure (Superseded by FILE_SAVEPOSITION_ Procedure)

Summary

The SAVEPOSITION procedure is used to save a disk file's current file positioning information in anticipation of a need to return to that position. The positioning information is returned to the file system in a call to the REPOSITION procedure when you want to return to the saved position.

Syntax for C Programmers

```
#include <cextdecs(SAVEPOSITION)>

_cc_status SAVEPOSITION ( short filenum
                        ,short _near *positioning-block
                        ,[ short _near *positioning-blksize ]
);
```

- The function value returned by SAVEPOSITION, which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL SAVEPOSITION ( filenum                          ! i
                  ,positioning-block                 ! o
                  ,[ positioning-blksize ] );        ! o
```

Parameters

filenum input

INT:value

is a number of an open file, identifying the file whose positioning block is to be obtained.

positioning-block output

INT:ref:*

returns the positioning information for the file's current position. The buffer must be large enough to hold the entire block of information. These shows how to calculate the required buffer size in words. (The maximum value for alternate key length is used to assure that the buffer is always large enough, although the system might return shorter blocks in specific cases.)

For key-sequenced files where positioning is performed by:

- Primary key, the count is calculated by

  $7 + (primary\text{-}keylen + 1) / 2$

- An alternate key, the count is calculated by

  $7 + (max\text{-}alternate\text{-}keylen + primary\text{-}keylen + 1) / 2$

For unstructured files and for entry-sequenced and relative files where no alternate keys exist, the count is 4. For entry-sequenced and relative files with alternate keys, where positioning is performed by:

- Primary key, the count is 7
- An alternate key, the count is calculated by

  $7 + (max\text{-}alternate\text{-}keylen + 4 + 1) / 2$

positioning-blksize output

INT:ref:1

returns the actual number of words in the positioning block.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILE_GETINFO_ or FILEINFO).

= (CCE) indicates that SAVEPOSITION is successful.

> (CCG) indicates that the file is not a disk file.

Considerations

- For relative and entry-sequence files that have no alternate keys. SAVEPOSITION requires a 7-word *positioning-block* if read-reverse is the current reading mode (see KEYPOSITION[X] Procedures (Superseded by FILE_SETKEY_ Procedure)).
- The SAVEPOSITION procedure cannot be used with Enscribe format 2 files or OSS files larger than approximately 2 gigabytes. If an attempt is made to use the SAVEPOSITION procedure with these files, error 581 is returned. For information on how to perform the equivalent task with Enscribe format 2 files or OSS files larger than approximately 2 gigabytes, see the FILE_SAVEPOSITION_ Procedure.
- In files that support insertion-ordered duplicate alternate keys, each alternate key includes four additional words for a timestamp.

Example

```
CALL SAVEPOSITION ( FILE^NUM , POSITION^BLOCK );
```

SEGMENT_ALLOCATE_ Procedure

Summary

The SEGMENT_ALLOCATE_ procedure allocates an extended data segment for use by the calling process. This procedure can create read/write segments, read-only segments, and extensible segments.

The SEGMENT_ALLOCATE_ procedure can also be used to share selectable extended data segments or flat extended data segments allocated by other processes (subject to the normal security requirements).

For selectable segments, the call to SEGMENT_ALLOCATE_ must be followed by a call to the SEGMENT_USE_ procedure to make the selectable extended data segment accessible. Although you can allocate multiple selectable segments, you can access only one at a time. For flat segments, the call to SEGMENT_ALLOCATE_ can be followed by a call to SEGMENT_USE_, but calling SEGMENT_USE_ is unnecessary

because all the flat segments allocated by a process are always accessible to the process.

Flat segments and selectable segments are supported on native processors that use D30 or later versions of the HP NonStop operating system. Selectable segments are supported on all systems.

Syntax for C Programmers

Note. In the TNS/E environment the CEXTDECS file uses the *int* data type for 32-bit values. This is a change from the TNS and TNS/R environments where CEXTDECS uses the *long* data type for 32-bit values.

```
#include <cextdecs(SEGMENT_ALLOCATE_)>

short SEGMENT_ALLOCATE_ ( short segment-id          /* i 1 */
                    ,[ __int32_t segment-size ]     /* i 2 */
                         ,[ char *filename ]      /* i:i 3 */
                         ,[ short length ]        /* i:i 3 */
                         ,[ short *error-detail ]   /* o 4 */
                         ,[ short pin ]             /* i 5 */
                         ,[ short segment-type ]    /* i 6 */
                    ,[ __int32_t *base-address ]] /* i,o 7 */
                    ,[ __int32_t max-size ]         /* i 8 */
                          ,[ short alloc-options ]););/*i 9*/
```

- The parameter *length* specifies the length in bytes of the character string pointed to by *filename*. The parameters *filename* and *length* must either both be supplied or both be absent.

Syntax for TAL Programmers

```
error := SEGMENT_ALLOCATE_ ( segment-id             ! i 1 !
                            ,[ segment-size ]       ! i 2 !
                            ,[ filename:length ]    ! i:i 3 !
                            ,[ error-detail ]       ! o 4 !
                            ,[ pin ]                ! i 5 !
                            ,[ segment-type ]       ! i 6 !
                            ,[ base-address ]       ! i,o,7 !
                            ,[ max-size ]           ! i 8 !
                            ,[ alloc-options ] );     ! i 9 !
```

Parameters

error returned value

INT

indicates the outcome of the operation. It contains one of these values:

0 No error
1 File-system error related to the creation or open of *filename*; *error-detail* contains the file-system error number.
2 Parameter error; *error-detail* contains the number of the first parameter found to be in error, where 1 designates the first parameter on the left.
3 Bounds error; *error-detail* contains the number of the first parameter found to be in error, where 1 designates the first parameter on the left.
4 Invalid *segment-id*
5 Invalid *segment-size*
6 Unable to allocate segment space
7 Unable to allocate segment page table space
8 Security violation when attempting to share segment
9 *pin* does not exist.
10 *pin* does not have the segment allocated.
11 Caller is trying to share segment with self.
12 Indicates one of three conditions: (1) The requested segment is a shared selectable segment, but the allocated segment is a flat segment. (2) The requested segment is a shared flat segment, but the allocated segment is a selectable segment. (3) The segment is being resized.
13 The *segment-id* parameter is already allocated by this process.
14 Unable to allocate process segment table (PST); *error-detail* contains the file-system error number.
15 Part or all of the requested address range has already been allocated. This error is returned if bit 15 of the *alloc-options* parameter is set to 1 and a flat segment cannot be allocated. This error can also occur when bit 15 is not set, but either a flat segment cannot be shared due to address-range overlap with another segment or a flat segment cannot be allocated as there is no unallocated address range large enough to hold the requested size. This error is returned only on native processors.

segment-id input

INT:value

is the number by which the process chooses to refer to the segment. Segment IDs are in the range 0 through 1023 for user processes; other values are reserved for processes supplied by HP. A nonprivileged process cannot supply a segment ID greater than 2047. Nonprivileged segment IDs are allocated as unaliased segments.

segment-size input

INT(32):value

specifies the size in bytes of the segment to be allocated.

- Flat segment size:
  - For G04.00 and earlier G-series RVUs, the value must be in the range 1 byte through 128 megabytes (134,217,728 bytes). A flat segment is allocated beginning on a 32-megabyte region boundary and is allocated

from a total virtual space of 480 megabytes (15 regions * 32 megabytes/region).

- For G05.00 and later G-series RVUs, the flat segment size limit is 1120 megabytes, theoretically. However, in a native mode process, the address space used for flat segments in C / C++ applications is also used for the heap. Flat segments, when allocated by Guardian, are assigned starting at the highest address and going downward, whereas the heap starts at the lowest address and grows upward. Therefore, for native mode programs, the maximum segment size is not actually 1120 MB. The maximum allowed depends on how much heap space the program uses. At best, a native mode program has 1119 MB available for flat segments, and it could have less available if the heap has grown to greater than 1 MB. An attempt to allocate an 1120 MB segment in a native program results in an error 15.
- The 32-megabyte region boundary does not apply in the G05.00 and later G-series RVUs.

- For a selectable segment, the value must be in the range 1 byte through 127.5 megabytes (133,693,440 bytes).

The system might round the size up to the next *segment-size* increment, where the increment is both processor-dependent and subject to change. The only effect this has on the program is that an address reference that falls outside the specified segment size but within the actual size does not cause an invalid address reference (trap 0 for a Guardian TNS process, a SIGSEGV signal for an OSS or any native process and a subsequent fetch might not retrieve the value previously stored).

For methods of sharing segments, see the *pin* and *segment-type* parameters.

Upon initial allocation of the segment:

- The *segment-size* parameter is required if the swap file does not exist.
- The *segment-size* parameter is optional if the swap file already exists. If the segment is a read-only segment, the default size is the end-of-file value of the swap file (EOF). If the segment is a read-write segment, the default segment size is the allocated size of the swap file.
- For a read-only segment, *segment-size* must not be greater than the end-of-file value of the file; otherwise, an error occurs. For a read-write segment, if *segment-size* is greater than the allocated size of the swap file, the system attempts to allocate additional space.

If a segment is being shared by the PIN method (see *pin*), this rule applies to the sharers:

- The *segment-size* parameter must be omitted, and the size of the segment is the same as that from the initial SEGMENT_ALLOCATE_ call.

If a segment is being shared by the file name method (see *segment-type*), these rules apply to the sharers:

- If the swap file supplied in the filename parameter does not exist at the time of the call, the segment-size parameter must be supplied.
- If the swap file supplied in the filename parameter exists, then:
  - The segment-size parameter can be omitted.
  - If the segment is a read-write segment, the default size is the allocated size of the swap file.
  - If the segment is a read-only segment, then:
    - If the swap file has not been opened in this processor by a previous call to SEGMENT_ALLOCATE_, the size of the segment will be set to segment-size, if supplied. If segment-size was not supplied, the size defaults to the size of the existing swap file.
    - If the swap file has already been opened in this processor by a prior call to SEGMENT_ALLOCATE_, the size of the segment will be set to the same size as the initally opened swap file, whether segment-size is supplied or not. (However, if segment-size is larger than the size specified in the original call to SEGMENT_ALLOCATE_, an error is returned.)

filename:*length* input:input

STRING .EXT:ref:*, INT:value

indicates several types of swap files: temporary swap space using KMSF, temporary swap file, existing permanent swap file, new permanent swap file, and segment sharing by the file-name method. If *filename* is specified, *pin* must be omitted.

- Temporary swap space using KMSF

  If you do not specify *filename* or if you specify *length* as 0 (and if a segment is not being shared using the PIN method), SEGMENT_ALLOCATE_ uses the Kernel-Managed Swap Facility (KMSF) to allocate swap space.

  Applications that share segments by the file-name method cannot be put under KMSF unless the sharing is changed to use the PIN. If you need to share by the file-name method, it will be necessary to use a temporary swap file (see “Temporary swap file” below) or to pass a volume name where the swap file should be created.

  Performance is increased by using KMSF. However, if you want to save the data in the segment after the process terminates, specify a permanent swap file name. KMSF swap files have the clear-on-purge attribute, which provides a level of security for swapped data.

For more information on KMSF, see the *Kernel-Managed Swap Facility (KMSF) Manual*.

- Temporary swap file

  If supplied, if *length* is not 0, and if *filename* is a volume name without a subvolume or file identifier, SEGMENT_ALLOCATE_ creates a temporary swap file on the indicated volume. If you specify a system name, it must be the system name of the local node. You can convert a temporary file to a permanent file by renaming it with the FILE_RENAME_ procedure.

  If you do not specify *filename* and bit 13 of the *alloc-options* parameter is set to 1, SEGMENT_ALLOCATE_ creates a temporary swap file on a volume that it chooses.

- Existing permanent swap file

  If supplied and if *length* is not 0, specifies the name of a swap file to be associated with the segment. If used, the value of *filename* must be exactly *length* bytes long. If the file name is partially qualified (for example, without the volume name), it is resolved using the contents of the =_DEFAULTS DEFINE. All data in the file is used as initial data for the segment. Remote file names, structured files, audited files, and files with the refresh attribute are not accepted.

  There are two advantages of using an existing swap file. First, if the file is the required size, segment allocation cannot fail due to lack of disk space. Second, the segment becomes a permanent repository of data.

  If the process terminates without deallocating the segment, any data still in memory is written back out to the file. Unless the segment is extensible, SEGMENT_ALLOCATE_ must be able to allocate a sufficient number of file extents to contain all memory in the segment.

- New permanent swap file

  If supplied, if *length* is not 0, and if *filename* does not exist, specifies the name of a swap file to be created. If used, the value of *filename* must be exactly *length* bytes long. Remote file names are not accepted.

  The advantage of using a permanent swap file is that the segment becomes a permanent repository of data.

  If the process terminates without deallocating the segment, any data still in memory is written back out to the file. Unless the segment is extensible, SEGMENT_ALLOCATE_ must be able to allocate a sufficient number of file extents to contain all memory in the segment.

- Segment sharing by the file name method

  By specifying *filename*, you can share the segment associated with this swap file with another process using the same swap file (provided that both processes have appropriate permission to the file). This is referred to as

segment sharing by the file-name method. Two processes sharing a segment by the file-name method must be in the same processor, unless the segment is a read-only segment. (See Considerations on page 14-14).

error-detail output

INT .EXT:ref:1

for some returned errors, contains additional information. See *error*.

pin input

INT:value

if present and not equal to -1, requests allocation of a segment that is shared by the PIN method. *pin* specifies the process identification number (PIN) of the process that has previously allocated the segment and with which the caller wants to share the segment. The process designated by *pin* must be in the same processor as the caller. Processes sharing a segment by this method must reference the segment by the same *segment-id*.

If *pin* is specified, *filename* must be omitted.

segment-type input

INT:value

describes the attributes of the segment to be allocated:

- Read-only

  A read-only segment is an extended data segment that is initialized from a preexisting swap file and used only for read access. A read-only segment can be shared by either the PIN or file-name method. It can also be shared by file name between processes in different processors. Note that the *filename* parameter must specify the name of an existing swap file that is not empty. Extensible read-only segments are not supported.

- Shared by file name

  The *filename* parameter must be supplied when this type of shared segment is allocated. Processes sharing segments by the file-name method can refer to the address space by different segment IDs and can supply different values for the segment size to SEGMENT_ALLOCATE_. The segment size supplied by the first allocator of a particular shared segment (as identified by the swap file name) establishes the upper limit for the segment size that can be set by processes subsequently attempting to share the segment.

  Callers that request sharing by file name must not supply the *pin* parameter.

- Extensible

  An extensible segment is an extended data segment for which the underlying swap file disk space is not allocated until needed. In this case, *segment-*

size is taken as a maximum size and the underlying virtual memory is expanded dynamically as the user accesses addresses within the extended data segment. When the user first accesses a portion of an extensible segment for which the corresponding swap file extent has not been allocated, the operating system allocates the extent. If the extent cannot be allocated, the user process terminates: a TNS Guardian process terminates with a “no memory available” trap (trap 12); an OSS or native process receives a SIGNOMEM signal.

If *segment-type* is omitted, the default value is 0. Valid values are:

0 Allocate a segment; sharing by the PIN method can be specified.
1 Allocate an extensible segment; sharing by the PIN method can be specified.
2 Allocate a segment with sharing by the file name method.
3 Allocate an extensible segment with sharing by the file name method.
4, 6 Allocate a read-only segment; sharing by either the PIN method or the file name method can be specified.

base-address input,output

EXTADDR .EXT:ref:1

if used as an output parameter, returns the base address of the segment being allocated. The *base-address* parameter can be used to determine whether the segment allocated is a flat segment or a selectable segment. The base address of a segment also can be obtained by calling the SEGMENT_GETINFO_ procedure.

- For a flat segment, the *base-address* output parameter value is different for each allocated segment. A flat segment is requested by setting bit 14 of the *alloc-options* parameter to 1.
- For a selectable segment, the *base-address* output parameter value is always %2000000D (%H00080000%D).

If *base-address* is used as an input parameter, it specifies the base address of the flat segment being allocated. This parameter is not updated on output. Bits 14 and 15 of the *alloc-options* parameter must be 1; otherwise, the value in the *base-address* parameter is ignored.

A program should usually allow the SEGMENT_ALLOCATE_ procedure to designate the address where a flat segment should start. In particular, library procedures that allocate flat segments should not specify a base address, because this allocation may be incompatible with other library-allocated or user-allocated segments within the same process. This feature can be useful for process pairs. For example, the primary process uses the *base-address* parameter as an output parameter and supplies the address to its backup process. The backup process, in turn, uses the *base-address* parameter as an input parameter to allocate the segment in the same place.

- For a flat segment, use the *base-address* input parameter only if it is necessary to force segment allocation to begin at a specific address. For

existing systems that support flat segments, the specified address must be a multiple of 32 kilobytes at or above %H30000000. Avoid hard-coding the address, because the valid range of addresses can change from RVU to RVU. An error is returned if the address is out of range, if the address is not a multiple of 32 kilobytes, or if the allocated segment would overlap a previously allocated segment.

- For a shared flat segment, the *base-address* input parameter maps the shared segment starting at the base specified address. If *base-address* is omitted, SEGMENT_ALLOCATE_ attempts to map the segment at the same base address as in the process that first allocated the segment. If that process no longer shares the segment, the default address is taken from one of the processes that still shares the segment. The SEGMENT_ALLOCATE_ call in the sharer will fail with error 15 in these cases:

  1. Another segment in the sharer is already mapped at this base-address, OR
  2. The address range of the segment to be shared overlaps with that of another segment in the sharer.

- For a selectable segment, the *base-address* input parameter is ignored, because the base address assigned to a selectable segment is always the same.

max-size input

INT(32):value

defines the upper limit of the *new-segment-size* parameter of the RESIZESEGMENT procedure. The value for *max-size* must be greater than or equal to the *segment-size* parameter and must be within the same range as the *segment-size* parameter.

alloc-options input

INT:value

provides information about the segment to be allocated. The bits are defined as follows:

<0:8>		Reserved (specify 0).
<9>	= 0	Map the segment at the same base address as the process that first allocated the segment.
	= 1	Same as for 0, but also, if the range of the requested segment is partially or completely overlapped in the current process, allocate the segment at any address within the flat segment space.
<10:12>		Reserved
<13>	= 0	If you do not specify *filename*, manage the swap space using the Kernel-Managed Swap Facility (KMSF).

	= 1	If you do not specify *filename*, create a temporary swap file.
<14>	= 0	Allocate a selectable segment.
	= 1	Allocate a flat segment.
<15>	= 0	Return the base address in the *base-address* parameter.
	= 1	Allocate a flat segment starting at the address specified in the *base-address* parameter. Bit 14 must also be set to 1.

The default value of *alloc-options* is all bits equal to 0, which means that a selectable segment is allocated and the base address is returned in the *base-address* parameter.

Considerations

Note. There are additional considerations for privileged callers.

- Preventing automatic temporary file purge

  SEGMENT_ALLOCATE_ opens the swap file for read/write/protected access. A process can prevent the automatic file purge of a temporary swap file by opening the file for read-only/shared access before the segment is deallocated.

- Nonexisting temporary swap file

  A caller requesting allocation of a temporary swap file can obtain the actual file name returned by making a subsequent call to SEGMENT_GETINFO_.

- Swap file extent allocation

  If an extensible segment is being created, then only one extent of the swap file is allocated when SEGMENT_ALLOCATE_ returns.

- Segment sharing

  Subject to security requirements, a process can share a segment with another process running on the same processor. For example, process $X can share a segment with any of these processes on the same processor:

  - Any process that has the same process access ID (PAID)
  - Any process that has the same group ID, if $X is the group manager (that is, if $X has a PAID of *group*,255)
  - Any process, if $X has a PAID of the super ID (255,255)

  If processes are running in different processors, they can share a segment only if the security requirements are met and the segment is a read-only segment. To specify a read-only segment, set bit 2 of the *segment-type* parameter.

Callers of ALLOCATESEGMENT can share segments with callers of SEGMENT_ALLOCATE_. High-PIN callers can share segments with low-PIN callers.

- Segment sharing by the file-name method

  In segment sharing by the file-name method, a read-write segment cannot be shared with read-only access.

- Sharing flat segments

  A process cannot share a flat segment with a process that allocated a selectable segment, because the segments reside in different parts of memory. (Similarly, a process cannot share a selectable segment with a process that allocated a flat segment.)

- Flat segments and increased performance

  Although the SEGMENT_USE_ and MOVEX procedures can be used with flat segments, you can improve performance by eliminating SEGMENT_USE_ calls and replacing MOVEX calls with direct assignment statements. Programs can determine the type of segment allocated and take advantage of the flat segment features whenever a flat segment is allocated.

- Selectable segments and performance

  If you have more than one selectable segment, you might face performance degradation, because time is wasted when switching between the selectable segments. This is because only one selectable segment is visible at a time. Instead, use flat segments, which are always visible.

- Determining whether a flat segment is allocated

  Use these techniques to determine whether a segment obtained is a flat segment:

  - Check the value returned in the *base-address* parameter. If the segment is a selectable segment, the base address is always %2000000D (%H00080000); otherwise, the segment is a flat segment. The *base-address* parameter of the SEGMENT_USE_ procedure also returns the base address of a segment.
  - Use the *usage-flags* parameter of the SEGMENT_GETINFO_ procedure.

  In addition, you can use the PROCESSOR_GETINFOLIST_ procedure (memory management attribute 57) to determine whether flat segments are supported on the processor. Flat segments are supported on native processors that use D30 or later versions of the NonStop operating system.

- Flat segments and user libraries

  A user library cannot maintain its own private global variables, so it has no way to retain the address of a flat segment allocated for its private use. A library can use a fixed segment ID when allocating a segment and then determine the base address in subsequent invocations by passing the same segment ID to the

SEGMENT_USE_ procedure. Alternatively, a library can return the base address to its client and have this address returned as a parameter in each library call.

- How segment space is divided

  To satisfy various hardware and software constraints and requirements for finding addresses within the virtual memory space, the space is divided into regions, segments, and pages.

Unit	Contains	Size of Unit
1 TNS page	2048 bytes	2048 bytes
1 unitary segment	64 TNS pages	64*2048 bytes = 128 kilobytes
1 Native page	process dependent	4096 or 16384 bytes
1 Native region	256 segments	256*128 kilobytes = 32 megabytes
1 gigabyte	32 regions	32*32 megabytes = 1024 megabytes
2 gigabytes	64 regions	64*32 megabytes = 2048 megabytes

Examples

```
error := SEGMENT_ALLOCATE_ ( segment^id, seg^size );
          ! standard call to create a user segment

error := SEGMENT_ALLOCATE_ ( segment^id, , , error^detail,
                             pin );
          ! allocates a shared segment, using the PIN
          ! method, which is shared with the segment given by
          ! segment^id in the process identified by pin.

error := SEGMENT_ALLOCATE_ ( segment^id, , filename:length,
                             error^detail, , 2 );
          ! allocates a shared segment using the filename
          ! method
```

Related Programming Manual

For programming information about the SEGMENT_ALLOCATE_ procedure, see the *Guardian Programmer's Guide*.

SEGMENT_ALLOCATE_CHKPT_ Procedure

Summary

The SEGMENT_ALLOCATE_CHKPT_ procedure is called by a primary process to allocate an extended data segment for use by its backup process.

Syntax for C Programmers

This passive backup procedure is not supported in C programs. For a comparison of active backup and passive backup, see the *Guardian Programmer's Guide*.

Syntax for TAL Programmers

```
error:= SEGMENT_ALLOCATE_CHKPT_ ( segment-id                ! i
                                 ,[ filename:length ]       !
i:i
                                 ,[ error-detail ]          ! o
                                 ,[ pin ] );                ! i
```

Parameters

error returned value

INT

indicates the outcome of the backup process's call to SEGMENT_ALLOCATE_. Any error returned to the backup by SEGMENT_ALLOCATE_ is returned here. For a list of possible values, see the definition of *error* under SEGMENT_ALLOCATE_.

segment-id input

INT:value

is the number by which the process chooses to refer to the segment. Segment IDs are in the range of 0 to 1023 for user processes; other values are reserved for processes supplied by HP. A nonprivileged process cannot supply a segment ID greater than 2047.

filename:*length* input:input

STRING .EXT:ref:*, INT:value

indicates several types of swap files: temporary swap space using KMSF, temporary swap file, existing permanent swap file, new permanent swap file, and segment sharing by the file-name method

If *filename* is specified, *pin* must be omitted.

- Temporary swap space using KMSF

  If you do not specify *filename* or if you specify *length* as 0 (and if a segment is not being shared using the PIN method), SEGMENT_ALLOCATE_CHKPT_ uses the Kernel-Managed Swap Facility (KMSF) to allocate swap space.

  To share this segment, use the PIN method; you cannot use the file-name method.

  Performance is increased by using KMSF. However, if you want to save the data in the segment after the process terminates, specify a permanent swap file name. KMSF swap files have the clear-on-purge attribute, which provides a level of security for swapped data.

  For more information on this facility, see the *Kernel-Managed Swap Facility (KMSF) Manual*.

- Temporary swap file

  If supplied, if *length* is not 0, and if *filename* is a volume name without a subvolume or file identifier, SEGMENT_ALLOCATE_CHKPT_ creates a temporary swap file on the indicated volume. If you specify a system name, it must be the system name of the local node. You can convert a temporary file to a permanent file by renaming it with the FILE_RENAME_ procedure.

  If you do not specify *filename* and bit 13 of the *alloc-options* parameter is set to 1, SEGMENT_ALLOCATE_CHKPT_ creates a temporary swap file on a volume that it chooses.

- Existing permanent swap file

  If supplied and if *length* is not 0, specifies the name of a swap file to be associated with the segment. If used, the value of *filename* must be exactly *length* bytes long. If the file name is partially qualified (for example, without the volume name), it is resolved using the contents of the =_DEFAULTS DEFINE. All data in the file is used as initial data for the segment. Remote file names, structured files, audited files, and files with the refresh attribute are not accepted.

  There are two advantages of using an existing swap file. First, if the file is the required size, segment allocation cannot fail due to lack of disk space. Second, the segment becomes a permanent repository of data.

If the process terminates without deallocating the segment, any data still in memory is written back out to the file. Unless the segment is extensible, SEGMENT_ALLOCATE_CHKPT_ must be able to allocate a sufficient number of file extents to contain all memory in the segment.

- New permanent swap file

  If supplied, if *length* is not 0, and if *filename* does not exist, specifies the name of a swap file to be created. If used, the value of *filename* must be exactly *length* bytes long. Remote file names are not accepted.

  The advantage of using a permanent swap file is that the segment becomes a permanent repository of data.

  If the process terminates without deallocating the segment, any data still in memory is written back out to the file. Unless the segment is extensible, SEGMENT_ALLOCATE_CHKPT_ must be able to allocate a sufficient number of file extents to contain all memory in the segment.

- Segment sharing by the file name method

  By specifying *filename*, you can share the segment associated with this swap file with another process using the same swap file (provided that both processes have appropriate permission to the file). This is referred to as segment sharing by the file-name method. Two processes sharing a segment by the file-name method must be in the same processor, unless the segment is a read-only segment. (For SEGMENT_ALLOCATE_, see Considerations on page 14-14.)

error-detail output

INT .EXT:ref:1

returns the error detail information returned from the backup process's call to SEGMENT_ALLOCATE_, or one of these file-system errors:

2	=	Segment is not allocated by the primary or the segment ID is invalid.
30	=	No message-system control blocks are available.
31	=	There is no room in the process file segment (PFS) for a message buffer in either the backup or the primary.
201	=	Unable to send to the backup

For information about the *error* values for which detail information can be returned here, see the description of the *error* parameter under the SEGMENT_ALLOCATE_ procedure.

pin input

INT:value

if present and not -1, requests allocation of a segment that is shared by the PIN method. *pin* specifies the process identification number of the process that has previously allocated the segment and with which the caller wishes to share the segment. The process designated by *pin* must be in the same processor as the caller. Processes sharing a segment by this method must reference the segment by the same *segment-id*.

If *pin* is specified, *filename* must be omitted.

Considerations

- The *segment-size*, *segment-type*, *max-size* and *alloc-options* parameters of SEGMENT_ALLOCATE_ are not supported in SEGMENT_ALLOCATE_CHKPT_ because the values for the primary process's segment are used.
- A segment with the same segment ID must be allocated in the primary process before the call to SEGMENT_ALLOCATE_CHKPT_.
- If the *filename* parameter is provided, that file name is used by SEGMENT_ALLOCATE_ in the backup process; otherwise, no file name parameter is passed to SEGMENT_ALLOCATE_ in the backup process.
- If you use the *pin* parameter, set it carefully because the PIN might not be the same on the backup processor. You must determine the correct PIN for the backup processor.
- If the segment is not read-only, the swap file name must be different on the backup and primary processors. If the same file name is given, allocation in the backup fails because swap files cannot be shared between processors.
- Nonexisting temporary swap file

  If a shared segment is being allocated and only a volume name is supplied in the filename parameter, then the complete file name of the temporary file created by SEGMENT_ALLOCATE_CHKPT_ can be obtained from a subsequent call to SEGMENT_GETBACKUPINFO_.
- Swap file extent allocation

  If an extensible segment is being created, then only one extent of the swap file is allocated when SEGMENT_ALLOCATE_CHKPT_ returns.
- Segment sharing

  Subject to security requirements, a process can share a segment with another process running on the same processor. For example, process $X can share a segment with any of these processes on the same processor:

- Any process that has the same process access ID (PAID)
- Any process that has the same group ID, if $X is the group manager (that is, if $X has a PAID of *group*,255)
- Any process, if $X has a PAID of the super ID (255,255)

If processes are running in different processors, they can share a segment only if the security requirements are met and the segment is a read-only segment.

Callers of [CHECK]ALLOCATESEGMENT can share segments with callers of SEGMENT_ALLOCATE_[CHKPT_]. High-PIN callers can share segments with low-PIN callers.

- Sharing flat segments

  A process cannot share a flat segment with a process that allocated a selectable segment, because the segments reside in different parts of memory. (Similarly, a process cannot share a selectable segment with a process that allocated a flat segment.)

SEGMENT_DEALLOCATE_ Procedure

Summary

The SEGMENT_DEALLOCATE_ procedure deallocates an extended data segment.

Syntax for C Programmers

```
#include <cextdecs(SEGMENT_DEALLOCATE_)>

short SEGMENT_DEALLOCATE_ ( short segment-id
                           ,[ short flags ]
                           ,[ short *error-detail ] );
```

Syntax for TAL Programmers

```
error := SEGMENT_DEALLOCATE_ ( segment-id           ! i
                              ,[ flags ]            ! i
                              ,[ error-detail ] );  ! o
```

Parameters

error returned value

INT

indicates the outcome of the operation. It returns one of these values:

0 Segment successfully deallocated.

1 Segment deallocated, but an I/O error occurred when writing to the segment's permanent swap file; *error-detail* contains the file-system error number.

2 Parameter error; *error-detail* contains the number of the first parameter found to be in error, where 1 designates the first parameter on the left.

3 Bounds error

4 Segment not deallocated; *error-detail* contains the reason for failure.

segment-id input

INT:value

specifies the segment ID of the segment to be deallocated. The segment ID was assigned by the program in the call to SEGMENT_ALLOCATE_ that allocated the segment.

flags input

INT:value

specifies whether dirty pages must be written to the swap file. A dirty page is a page in memory that has been updated but not written to the swap file. Valid values are:

<0:14> Reserved (specify 0)

<15> 1 Indicates that dirty pages in memory are not to be written to the swap file.

0 Indicates that dirty pages in memory are to be written to the swap file.

This parameter is ignored if the swap space was allocated using the Kernel-Managed Swap Facility (KMSF).

The default is 0.

error-detail output

INT .EXT:ref:1

returns additional information associated with some errors. For details, see *error*.

If *error* is 4, *error-detail* returns one of these values:

1 *segment-id* is out of range.

2 *segment-id* is in range but not allocated by the caller.

3 Segment is currently in use by the system. It might be in this state because an outstanding nowait I/O operation using a buffer within the segment has not been completed by a call to AWAITIOX.

30 No message-system control blocks are available.

Considerations

- *flags* parameter

  If the swap file associated with an extended data segment is neither a temporary file nor managed by the Kernel-Managed Swap Facility (KMSF), all of the modified pages of the segment are written to the file before it is closed by the last process using it. This is also true for a swap file that was created as a temporary file but was later renamed. (A program might use this method to keep its temporary file.) However, if the extended segment is large and if there are a large number of modified (“dirty”) pages, it might take a long time to deallocate the segment. If *flags*.<15> is set to 1, the modified pages are *not* written to the swap file, even if it is a permanent file. This option is recommended when the swap file has been made permanent to reserve the swap file space, or when the file contents are unimportant for any reason.

- Breakpoints

  Before deallocating a segment, SEGMENT_DEALLOCATE_ removes all memory access breakpoints set in that segment.

- Segment deallocation

  When a segment is deallocated, the swap file end of file (EOF) is set to the larger of these values:

- the EOF when the file was opened by SEGMENT_ALLOCATE_
- the end of the highest numbered page that is written to the swap file

- Shared segments

  A shared segment remains in existence until it has been deallocated by all the processes that allocated it.

Example

```
error := SEGMENT_DEALLOCATE_ ( segment^id, , error^detail );
```

Related Programming Manual

For programming information about the SEGMENT_DEALLOCATE_ memory management procedure, see the *Guardian Programmer's Guide*.

SEGMENT_DEALLOCATE_CHKPT_ Procedure

Summary

The SEGMENT_DEALLOCATE_CHKPT_ procedure is called by a primary process to deallocate an extended data segment in its backup process.

Syntax for C Programmers

This passive backup procedure is not supported in C programs. For a comparison of active backup and passive backup, see the *Guardian Programmer's Guide*.

Syntax for TAL Programmers

```
error := SEGMENT_DEALLOCATE_CHKPT_ ( segment-id            ! i
                                   ,[ flags ]              ! i
                                   ,[ error-detail ] );    ! o
```

Parameters

error returned value

INT

indicates the outcome of the operation. It returns one of these values:

0 Segment successfully deallocated.

1 Segment deallocated, but an I/O error occurred when writing to the segment's permanent swap file; *error-detail* contains the file-system error number.

2 Parameter error; *error-detail* contains the number of the first parameter found to be in error, where 1 designates the first parameter on the left.

3 Bounds error

4 Segment not deallocated; *error-detail* contains the reason for failure.

segment-id input

INT:value

specifies the segment ID of the segment to be deallocated. The segment ID was assigned by the program in the call to SEGMENT_ALLOCATE_ that allocated the segment.

flags input

INT:value

specifies whether dirty pages must be written to the swap file. A dirty page is a page in memory that has been updated but not written to the swap file. Valid values are:

<0:14> Reserved (specify 0)

<15> 1 Indicates that dirty pages in memory are not to be written to the swap file.

0 Indicates that dirty pages in memory are to be written to the swap file.

The default is 0.

error-detail output

INT .EXT:ref:1

returns additional information associated with some errors. For details, see *error* .

If *error* is 4, *error-detail* returns one of these values:

1 *segment-id* is out of range.

2 *segment-id* is in range but not allocated by the caller.

3 Segment is currently in use by the system. It might be in this state because an outstanding nowait I/O operation using a buffer within the segment has not been completed by a call to AWAITIOX.

30 No message-system control blocks are available.

31 There is no room in the process file segment (PFS) for a message buffer in either the backup or the primary.

201 Unable to send to the backup.

Considerations

- The segment need not be allocated by the primary process at the time of the call to SEGMENT_DEALLOCATE_CHKPT_.
- *flags* parameter

  If the swap file associated with an extended data segment is not a temporary file, all of the modified pages of the segment are written to the file before it is closed by the last process using it. This is also true for a swap file that was created as a temporary file but was later renamed. (A program might use this method to keep its temporary file.) However, if the extended segment is large and if there are a large number of modified ("dirty") pages, it might take a long time to deallocate the file. If *flags*.<15> is set to 1, the modified pages are *not* written to the swap file, even if it is a permanent file. This option is recommended when the swap file has been made permanent to reserve the swap file space, or when the file contents are unimportant for any reason.
- Breakpoints

  Before deallocating a segment, SEGMENT_DEALLOCATE_CHKPT_ removes all memory access breakpoints set in that segment.
- Segment deallocation

  When a segment is deallocated, the swap file end of file (EOF) is set to the larger of these values:

  - the EOF when the file was opened by SEGMENT_ALLOCATE_
  - the end of the highest numbered page that is written to the swap file
- Shared segments

  A shared segment remains in existence until it has been deallocated by all the processes that allocated it.

SEGMENT_GETBACKUPINFO_ Procedure

Summary

The SEGMENT_GETBACKUPINFO_ procedure retrieves information about an extended segment that is allocated by the backup process in a named process pair.

Syntax for TAL Programmers

```
error := SEGMENT_GETBACKUPINFO_ ( segment-id              ! i
                                 ,[ segment-size ]        ! o
                                 ,[ filename:maxlen ]     ! o:i
                                 ,[ filename-len ]        ! o
                                 ,[ error-detail ]        ! o
                                 ,[ base-address ] );     ! o
```

Parameters

error returned value

INT

indicates the outcome of the operation. Returns one of these values:

0 No error

1 Error occurred when attempting to obtain *filename*; *error-detail* contains the file-system error number.

2 Parameter error; *error-detail* contains the number of the first parameter found to be in error, where 1 designates the first parameter on the left.

3 Bounds error; *error-detail* contains the number of the first parameter found to be in error, where 1 designates the first parameter on the left.

4 *segment-id* is out of range.

5 *segment-id* is in range but not allocated by caller.

6 Information not obtained; *error-detail* contains the reason for failure.

segment-id input

INT:value

specifies the segment ID of the extended segment for which information is to be returned.

segment-size output

INT(32) .EXT:ref:1

returns the current size in bytes of the specified segment. The size might be rounded up from what was specified when the segment was allocated. For details, see the description of *segment-size* under SEGMENT_ALLOCATE_ Procedure.

filename:*maxlen* output:input

STRING .EXT:ref:*, INT:value

if present and *maxlen* is not 0, returns the fully qualified name of the segment's swap file.

If the segment is managed by the Kernel-Managed Swap Facility (KMSF), *filename* is undefined. For more information on KMSF, see the *Kernel-Managed Swap Facility (KMSF) Manual*.

maxlen specifies the length in bytes of the string variable *filename*.

filename-len output

INT .EXT:ref:1

returns the length in bytes of the swap-file name being returned.

If the segment is managed by the Kernel-Managed Swap Facility (KMSF), *filename-len* is set to 0. For more information on KMSF see the *Kernel-Managed Swap Facility (KMSF) Manual*.

error-detail output

INT .EXT:ref:1

returns additional information associated with some errors.

if *error* is 6, one of these values is returned:

30 No message-system control blocks available

31 The process file segment (PFS) has no room for a message buffer in either the backup or the primary.

201 Unable to send to the backup

For other uses of *error-detail*., see information under *error*.

base-address output

INT(32) .EXT:ref:1

returns the base address of the segment :

- For a flat segment, *base-address* is different for each allocated segment.

- For a selectable segment, *base-address* is always %2000000D (%H00080000).

SEGMENT_GETINFO_ Procedure

Summary

The SEGMENT_GETINFO_ procedure retrieves information about a currently allocated extended data segment.

Syntax for C Programmers

Note. In the TNS/E environment the CEXTDECS file uses the *int* data type for 32-bit values. This is a change from the TNS and TNS/R environments where CEXTDECS uses the *long* data type for 32-bit values.

```
#include <cextdecs(SEGMENT_GETINFO_)>

short SEGMENT_GETINFO_ ( short segment-id
                        ,[ __int32_t *segment-size ]
                        ,[ char *filename ]
                        ,[ short maxlen ]
                        ,[ short *filename-len ]
                        ,[ short *error-detail ]
                        ,[ __int32_t *base-address ]
                        ,[ short *usage-flags ] );
```

- The parameter *maxlen* specifies the maximum length in bytes of the character string pointed to by *filename*, the actual length of which is returned by

filename-len. All three of these parameters must either be supplied or be absent.

Syntax for TAL Programmers

```
error := SEGMENT_GETINFO_ ( segment-id              ! i
                          ,[ segment-size ]         ! o
                          ,[ filename:maxlen ]      ! o:i
                          ,[ filename-len ]         ! o
                          ,[ error-detail ]         ! o
                          ,[ base-address ]         ! o
                          ,[ usage-flags ] );       ! o
```

Parameters

error returned value

INT

Indicates the outcome of the operation. Returns one of these values:

0 No error

1 Error occurred when attempting to obtain *filename*; *error-detail* contains the file-system error number.

2 Parameter error; *error-detail* contains the number of the first parameter found to be in error, where 1 designates the first parameter on the left.

3 Bounds error; *error-detail* contains the number of the first parameter found to be in error, where 1 designates the first parameter on the left.

4 *segment-id* is out of range.

5 *segment-id* is in range but not allocated by caller.

segment-id input

INT:value

specifies the segment ID of the extended data segment for which information is to be returned.

segment-size output

INT(32) .EXT:ref:1

returns the current size in bytes of the specified extended data segment. The size might be rounded up from what was specified when the segment was allocated. For details, see the description of the *segment-size* parameter of SEGMENT_ALLOCATE_ Procedure.

filename:*maxlen* output:input

STRING .EXT:ref:*, INT:value

if present and *maxlen* is not 0, returns the fully qualified name of the segment's swap file.

If the segment is managed by the Kernel-Managed Swap Facility (KMSF), *filename* is undefined. For more information on KMSF see the *Kernel-Managed Swap Facility (KMSF) Manual*.

maxlen specifies the length in bytes of the string variable *filename*.

filename-len output

INT .EXT:ref:1

returns the length in bytes of the swap-file name being returned.

If the segment is managed by the Kernel-Managed Swap Facility (KMSF), *filename-len* is set to 0. For more information on KMSF see the *Kernel-Managed Swap Facility (KMSF) Manual*.

error-detail output

INT .EXT:ref:1

returns additional information associated with some errors. For details, see *error*.

base-address output

EXTADDR .EXT:ref:1

returns the base address of the segment:

- For a flat segment, *base-address* is different for each allocated segment.
- For a selectable segment, *base-address* is always %2000000D (%H00080000).

usage-flags output

INT .EXT:ref:1

returns additional information about the extended data segment. The bits, when set to 1, indicate:

<0:5> (Bits are reserved; 0 is returned)

<6> Segment is managed by the Kernel-Managed Swap Facility (KMSF).

<7> Segment is an OSS shared memory segment.

<8> Segment is an unaliased segment. An unaliased segment does not have a corresponding absolute segment address.

<9> Segment is a flat segment.

<10> Segment is a resident cache segment.

<11> Segment can be shared.

<12> Segment is the currently in-use selectable segment for the process.

<13> Segment is read-only.

<14> Segment is extensible.

<15> Segment is resident.

Considerations

For the SEGMENT_ALLOCATE_ procedure, see Considerations on page 14-14.

Example

```
error := SEGMENT_GETINFO_ ( segment^id, seg^size,
                            swap^file:length );
```

Related Programming Manual

For programming information about the SEGMENT_GETINFO_ procedure, see the *Guardian Programmer's Guide*.

SEGMENT_USE_ Procedure

Summary

The SEGMENT_USE_ procedure selects a particular extended data segment to be currently addressable by the calling process.

For selectable segments, the call to SEGMENT_USE_ must follow a call to SEGMENT_ALLOCATE_ to make the selectable extended data segment accessible. Although you can allocate multiple selectable extended data segments, you can access only one at a time.

For flat segments, the call to SEGMENT_USE_ can follow a call to SEGMENT_ALLOCATE_, but calling SEGMENT_USE_ is unnecessary because all of the flat segments allocated by a process are always accessible to the process.

Flat segments and selectable segments are supported on native processors that use D30 or later versions of the NonStop operating system. Selectable segments are supported on all systems.

Syntax for C Programmers

Note. In the TNS/E environment the CEXTDECS file uses the *int* data type for 32-bit values. This is a change from the TNS and TNS/R environments where CEXTDECS uses the *long* data type for 32-bit values.

```
#include <cextdecs(SEGMENT_USE_)>

short SEGMENT_USE_ ( short new-segment-id
                    ,[ short *old-segment-id ]
                    ,[ __int32_t *base-address ]
                    ,[ short *error-detail ] );
```

Syntax for TAL Programmers

```
error := SEGMENT_USE_ ( new-segment-id          ! i
                       ,[ old-segment-id ]      ! o
                       ,[ base-address ]        ! o
                       ,[ error-detail ] );     ! o
```

Parameters

error returned value

INT

indicates the outcome of the operation. It returns one of these values:

0 No error; the requested values are returned.

2 Parameter error; *new-segment-id* parameter is missing.

3 Bounds error; *error-detail* contains the number of the first parameter found to be in error, where 1 designates the first parameter on the left. This error is returned only to nonprivileged callers.

4 *new-segment-id* is not allocated.

5 *new-segment-id* is out of range.

new-segment-id input

INT:value

if not -1, specifies the segment ID of the selectable extended data segment to be put into use. A value of -1 indicates that the current selectable extended data segment should be taken out of use and that no new segment should be put into use.

If *new-segment-id* specifies a flat segment, *old-segment-id* returns the segment ID of the current in-use selectable segment. The flat segment and the selectable segment remain addressable by the calling process.

old-segment-id output

INT .EXT:ref:1

returns the segment ID of the selectable extended data segment that was previously in use. If no selectable segment was in use, a value of -1 is returned.

If *new-segment-id* specifies a flat segment, *old-segment-id* returns the segment ID of the current in-use selectable segment. The flat segment and the selectable segment remain addressable by the calling process.

base-address output

EXTADDR .EXT:ref:1

returns the base address of the segment specified by *new-segment-id*:

- For a flat segment, *base-address* is different for each allocated segment.
- For a selectable segment, *base-address* is always %2000000D (%H00080000).

error-detail output

INT .EXT:ref:1

returns additional error information when an *error* value of 3 (bounds error) is returned. For details, see *error*.

Considerations

For the SEGMENT_ALLOCATE_ procedure, see Considerations on page 14-14.

Example

```
error := SEGMENT_USE_ ( new^seg^id, old^seg^id,
                        base^address, error^detail );
```

Related Programming Manual

For programming information about the SEGMENT_USE_ procedure, see the *Guardian Programmer's Guide*.

SEGMENTSIZE Procedure (Superseded by SEGMENT_GETBACKUPINFO_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The SEGMENTSIZE procedure returns the size of the specified segment in bytes.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
seg-size := SEGMENTSIZE ( segment-id );          ! i
```

Parameters

seg-size returned value

INT(32)

is the size of segment *segment-id* in bytes, or -1D as an error indication if *segment-id* is an invalid segment number or is a segment not accessible by the caller.

segment-id input

INT:value

is the segment ID number of a segment accessible by the calling process. If *segment-id* is an invalid segment number or is a segment not accessible by the caller, then SEGMENTSIZE returns -1D as an error indication.

Example

```
INT(32) seglen := 0D;
INT segid := 2;           ! pass to SEGMENTSIZE later
      .
      .
seglen := SEGMENTSIZE ( segid );
IF seglen = -1D THEN ...
```

SENDBREAKMESSAGE Procedure (Superseded by BREAKMESSAGE_SEND_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

SENDBREAKMESSAGE allows user processes to send break-on-device messages to other processes.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
error := SENDBREAKMESSAGE(    process-id        ! i
                          [ ,breaktag ] );     ! i
```

Parameters

error returned value

INT

is the file-system error number indicating the outcome of the call.

process-id input

INT .EXT:ref:4

identifies the process to which the break-on-device message is to be sent. The format of the 4-word process ID is:

[0:2]	Process name or creation timestamp
[3].<0:3>	Reserved
.<4:7>	Processor number where the process is executing
.<8:15>	PIN assigned by the operating system to identify the process in the processor

breaktag input

INT .EXT:ref:2

if present, is a user-defined value to be delivered in the fourth and fifth words of the break-on-device message. This value corresponds to the break tag value that can be supplied to an access method with SETPARAM function 3.

Considerations

- A successful status indication from SENDBREAKMESSAGE does not imply that the process has received the message, only that it has been sent.
- If the *process-id* designates a member of a named process pair, the break-on-device message delivery will automatically be retried to the backup process if a failure or path switch occurs.
- A break-on-device system message is delivered to the $RECEIVE file of the target process. For the format of the interprocess system messages, see the *Guardian Procedure Errors and Messages Manual*.
- SENDBREAKMESSAGE cannot be used for a high-PIN unnamed process because a high PIN cannot fit into *process-id*; BREAKMESSAGE_SEND_ should be used instead. However, it can use SENDBREAKMESSAGE for a high-PIN *named* process.

SET^FILE Procedure

Summary

The SET^FILE procedure alters file characteristics and checks the old values of those characteristics being altered.

SET^FILE is a sequential I/O (SIO) procedure and should be used only with files that have been opened by OPEN^FILE.

Syntax for C Programmers

For Native C programs:

```
#include <cextdecs(SET_FILE)>

short SET_FILE ( short { *common-fcb }
                       { *file-fcb   }
                ,short operation
                ,[ short new-value ]
                ,[ short *old-value ]
                ,[ short setaddr-value ] );
```

For C programs:

```
#include <cextdecs(SET_FILE)>

short SET_FILE ( short { *common-fcb }
                       { *file-fcb   }
                ,short operation
                ,[ short new-value ]
                ,[ short *old-value ] );
```

Syntax for TAL Programmers

For pTAL callers, the procedure definition is:

```
error := SET^FILE ( { common-fcb }            ! i
                    { file-fcb   }            ! i
                   ,operation                 ! i
                   ,[ new-value ]             ! i
                   ,[ old-value ]             ! o
                   ,[ setaddr-value ]);       ! i
```

For other callers, the procedure definition is:

```
error := SET^FILE ( { common-fcb }            ! i
                    { file-fcb   }            ! i
                   ,operation                 ! i
                   ,[ new-value ]             ! i
                   ,[ old-value ] );          ! o
```

Parameters

error returned value

INT

returns a file-system or sequential I/O (SIO) procedure error number indicating the outcome of the SET^FILE.

If abort-on-error mode is in effect, the only possible value for *error* is 0.

common-fcb input

INT:ref:*

identifies those files whose characteristics are to be altered.

The SET^FILE operations that make sense only for the *common-fcb* are the SET^BREAKHIT, SET^ERRORFILE, and SET^TRACEBACK.

When using INIT^FILEFCB, INIT^FILEFCB^D00, FILE^FWDLINKFCB, FILE^BWDLINKFCB, and SET^TRACEBACK the FCB can be specified as the *common-fcb* or the *file-fcb*.

If an improper FCB is specified or the FCB is not initialized, an error is indicated.

file-fcb input

INT:ref:*

identifies the file whose characteristics are to be altered. In most cases, the FCB must be associated with a file or $RECEIVE.

When using INIT^FILEFCB, INIT^FILEFCB^D00, FILE^FWDLINKFCB, FILE^BWDLINKFCB, and SET^TRACEBACK, the FCB can be specified as the *common-fcb* or the *file-fcb*.

If an improper FCB is specified or the FCB is not initialized, an error is indicated.

operation input

INT:value

specifies the file characteristic to be altered. (See Table 14-2 on page 14-41 and Table 14-3 on page 14-51.)

new-value input

INT:value

specifies a new value for the specified *operation*. This parameter is optional, depending on the operation desired. For pTAL callers, some operations require that the *setaddr-value* parameter be used.

old-value output

INT:ref:*

is a variable in which the current value for the specified *operation* returns. This can vary from 1 to 12 words and is useful in saving this value for reset later. If *old-value* is omitted, the current value is not returned.

setaddr-value input

WADDR:ref:*

for pTAL callers only, specifies a new address for the specified *operation*. This parameter is optional, depending on the operation desired.

Considerations

- Table 14-2 on page 14-41 contains operations that set values in the *new-value* parameter.
- Table 14-3 on page 14-51 contains operations that set addresses. For pTAL callers, addresses are set in the *setaddr-value* parameter. For other callers, addresses are set in the *new-value* parameter
- In Table 14-2 on page 14-41 and Table 14-3 on page 14-51, the column labeled “State of File” is flagged with:

  Open The file must be opened to alter the file’s characteristics.

  Closed The file must be closed to alter the file’s characteristics.

  Any The file can either be open or closed to alter the file’s characteristics.

- The ASSIGN^SECEXT operation comes into effect only when an ASSIGN^PRIEXT (or ASSIGN^PRIMARYEXTENTSIZE) operation is specified. If the primary extent (ASSIGN^PRIEXT) has not been set explicitly, any command to set the secondary extent (ASSIGN^SECEXT) will be ignored, and the secondary extent will have the default value. This table describes operations that set values in the *new-value* parameter.

.

Table 14-2. SET^FILE Operations That Set Values (page 1 of 10)

operation	**Description of Operation Requested**	*new-value*	*old-value*	**State of File**
ASSIGN^ BLOCKBUFLEN (or ASSIGN^ BLOCKLENGTH)	Specifies the block length, in bytes, for the file.	<new-blocklen>	<blocklen>	Closed
ASSIGN^FILECODE	Specifies the file code for the file.	<new-file-code>	<file-code>	Closed
ASSIGN^ OPENACCESS	Specifies the open access for the file. These literals are provided for <open-access>: READWRITE^ACCESS (0) READ^ACCESS (1) WRITE^ACCESS(2) Even if WRITE^ACCESS is specified, SIO actually opens the file with READWRITE^ACCESS to facilitate interactive I/O.	<new-open-access>	<open-access>	Closed

Table 14-2. SET^FILE Operations That Set Values (page 2 of 10)

operation	**Description of Operation Requested**	*new-value*	*old-value*	**State of File**
ASSIGN^ OPENEXCLUSION	Specifies the open exclusion for the file. These literals are provided for <open-exclusion>: SHARE(0) EXCLUSIVE (1) PROTECTED(3)	<new-open-exclusion>	<open-exclusion>	Closed
ASSIGN^PRIEXT (or ASSIGN^ PRIMARY EXTENTSIZE)	Specifies the primary extent size (in units of 2048-byte blocks) for the file.	<new-pret-ext-size>	<pri-ext-size>	Closed
ASSIGN^RECORDLEN (or ASSIGN^ RECORDLENGTH)	Specifies the logical record length (in bytes) for the file. ASSIGN^ RECORDLENGTH gives the default read or write count. For default values, see the *Guardian Programmer's Guide*.	<new-recordlen>	<recordlen>	Closed
ASSIGN^SECEXT (or ASSIGN^ SECONDARY EXTENTSIZE)	Specifies the secondary extent size (in units of 2048-byte blocks) for the file. When set alone without ASSIGN^PRIEXT, this operation sets only the default values; it comes into effect only when a ASSIGN^PRIEXT is set.	<new-sec-ext-size>	<sec-ext-size>	Closed

Table 14-2. SET^FILE Operations That Set Values (page 3 of 10)

operation	**Description of Operation Requested**	*new-value*	*old-value*	**State of File**
INIT^FILEFCB	Specifies that the file FCB be initialized. This operation is not used when the INITIALIZER procedure is called to initialize the FCBs. It is valid only for C-series format FCBs. For example: CALL SET^FILE (common^fcb, INIT^FILEFCB); CALL SET^FILE (in^file,INIT^FILEFCB);	must be omitted	must be omitted	Closed
INIT^FILEFCB^D00	Specifies that the file FCB be initialized. This operation is not used when the INITIALIZER procedure is called to initialize the FCB. It is valid only for D-series format FCBs. For example: CALL SET^FILE (common^fcb, INIT^FILEFCB^D00); CALL SET^FILE (in^file, INIT^FILEFCB^D00);	must be omitted	must be omitted	Closed

Table 14-2. SET^FILE Operations That Set Values (page 4 of 10)

operation	**Description of Operation Requested**	*new-value*	*old-value*	**State of File**
SET^ABORT^ XFERERR	Sets or clears abort-on-transfer error for the file. If on, and a fatal error occurs during a data-transfer operation (such as a call to any SIO procedure except OPEN^FILE), all files are closed and the process abnormally ends. If off, the file-system or SIO procedure error number returns to the caller.	<new-state>	<state>	Open
SET^BREAKHIT	Sets or clears break hit for the file. This is used only if the user is handling BREAK independently of the SIO procedures, or if the user has requested BREAK system messages through SET^SYSTEMMESSAGES or SET^SYSTEMMESSAGES MANY.	<new-state>	<state>	Any
SET^CHECKSUM	Sets or clears the checksum word in the FCB. This is useful after modifying an FCB directly (that is, without using the SIO procedures).	<new-checksum-word>	<checksum-word-in-fcb>	Any

Table 14-2. SET^FILE Operations That Set Values (page 5 of 10)

operation	**Description of Operation Requested**	*new-value*	*old-value*	**State of File**
SET^COUNTXFERRED	Sets the physical I/O count, in bytes, transferred for the file. This is used only if nowait I/O is in effect and the user is making the call to AWAITIO for the file. This is the <count-transferred> parameter value returned from AWAITIO.	<new-count>	<count>	Open
SET^CRLF^BREAK	Sets or clears carriage return/line feed (CR/LF) on BREAK for the file. If on, a CR/LF is executed on the terminal when the BREAK key is pressed.	<new-state>	<state>	Open
SET^EDITLINE^ INCREMENT	Specifies the EDIT line increment to be added to successive line numbers for lines that will be added to the file. The value should be specified as 1000 times the line number increment value. The default value is 1000, which corresponds to an increment of 1. The possible EDIT line numbers are from 0 to 99999.999	<new-increment>	<increment>	Open

Table 14-2. SET^FILE Operations That Set Values (page 6 of 10)

operation	**Description of Operation Requested**	*new-value*	*old-value*	**State of File**
SET^EDITREAD^ REPOSITION	Specifies that this READ^FILE is to begin at the position set in the sequential block buffer (second through fourth words). For example: CALL SET^FILE(EDIT^FCB, SET^EDITREAD^ REPOSITION); See discussion of the SET^EDITREAD^REPOSITION operation in the *Guardian Programmer's Guide*.	must be omitted	must be omitted	Open
SET^ERROR	Sets file-system error code value for the file. This is used only if nowait I/O is in effect and the user makes the call to AWAITIO for the file. This is the <error> parameter value returned from FILEINFO.	<new-error>	<error>	Open
SET^PHYSIOOUT	Sets or clears physical I/O outstanding for the file specified by <file-fcb>. This is used only if nowait I/O is in effect and the user makes the call to AWAITIO for the file.	<new-state>	<state>	Open
SET^PRINT^ERR^MSG	Sets or clears print error message for the file. If on and a fatal error occurs, an error message is displayed on the error file. This is the home terminal unless otherwise specified.	<new-state>	<state>	Open

Table 14-2. SET^FILE Operations That Set Values (page 7 of 10)

operation	**Description of Operation Requested**	*new-value*	*old-value*	**State of File**
SET^PROMPT	Sets interactive prompt for the file. See the OPEN^FILE procedure.	<new-prompt-char>	<prompt-char>	Open
SET^RCVEOF	Sets return end of file (EOF) on process close for $RECEIVE file. This causes an EOF indication to be returned from READ^FILE when the receive open count goes from 1 to 0. The setting for return EOF has no meaning if the user is monitoring open and close messages. If the file is opened with read-only access, the setting defaults to on for return EOF.	<new-state>	<state>	Open
SET^RCVOPENCNT	Sets receive open count for the $RECEIVE file. This operation is intended to clear the count of openers when an open already accepted by the SIO procedures is subsequently rejected by the user. See SET^RCVUSEROPENREPLY.	<new-receive-open-count>	<receive-open-count>	Open

Table 14-2. SET^FILE Operations That Set Values (page 8 of 10)

operation	**Description of Operation Requested**	*new-value*	*old-value*	**State of File**
SET^ RCVUSEROPENREPLY (*continued*)	If <state> is 0, a return from READ^FILE is made only when data is received. Note: If open message = 1 is specified to SET^SYSTEMMESSAGES or SET^SYSTEMMESSAGESMANY, the setting of SET^ RCVUSEROPENREPLY has no meaning. An <error> of 6 returns from READ^FILE if an open message is accepted by the SIO procedures.	<new-state>	<state>	Open
SET^READ^TRIM	Sets or clears read-trailing-blank-trim for the file. If on, the <count-read> parameter does not account for trailing blanks.	<new-state>	<state>	Open

Table 14-2. SET^FILE Operations That Set Values (page 9 of 10)

operation	**Description of Operation Requested**	*new-value*	*old-value*	**State of File**
SET^ SYSTEMMESSAGES (*continued*)	.<15> = unused The user replies to the system messages designated by this operation by using WRITE^FILE. If no WRITE^FILE is encountered before the next READ^FILE, a <reply-error-code> = 0 is made automatically. Note that this operation cannot set some of the newer system messages; for these, use SET^SYSTEMMESSAGESMANY.	<new-sys-msg-mask>	<sys-msg-mask>	Open
SET^TRACEBACK	Sets or clears the traceback feature. When traceback is active, the SIO facility appends the caller's P-relative address to all error messages.	<new-state>	<old-state>	Any
SET^USERFLAG	Sets user flag for the file. The user flag is a one-word value in the FCB that the user can manipulate to maintain information about the file.	<new-user-flag>	<user-flag-in-fcb>	Open

Table 14-2. SET^FILE Operations That Set Values (page 10 of 10)

operation	**Description of Operation Requested**	*new-value*	*old-value*	**State of File**
SET^WRITE^FOLD	Sets or clears write-fold for the file. If on, write^file operations exceeding the record length cause multiple logical records to be written. If off, write^file operations exceeding the record length are truncated to record-length bytes; no error message or warning is given.	<new-state>	<state>	Any
SET^WRITE^PAD	Sets or clears write-blank-pad for the file. If on, write^file operations of less than record-length bytes, including the last record if WRITE^FOLD is in effect, are padded with trailing blanks to fill out the logical record.	<new-state>	<state>	Open
SET^WRITE^TRIM	Sets or clears write-trailing-blank-trim for the file. If on, trailing blanks are trimmed from the output record before being written to the line.	<new-state>	<state>	Open

This table describes operations that set addresses. For pTAL callers, addresses are set in the *setaddr-value* parameter. For other callers, addresses are set in the *new-value* parameter.

Table 14-3. SET^FILE Operations That Set Addresses (page 1 of 5)

operation	**Description of Operation Requested**	*setaddr-value or new-value*	*old-value*	**State of File**
SET^ SYSTEMMESSAGES MANY	Sets system message reception for the $RECEIVE file. <sys-msg-mask-words> is a four-word mask. Setting a bit in <sys-msg-mask-words> indicates that the corresponding message is to pass back to the user. Default action is for the SIO procedures to handle all system messages.	@<new-sys-msg-mask-word>	<sys-msg-mask-words>	Open

Table 14-3. SET^FILE Operations That Set Addresses (page 2 of 5)

operation	**Description of Operation Requested**	*setaddr-value or new-value*	*old-value*	**State of File**
SET^ SYSTEMMESSAGES MANY (*word 0*)	<sys-msg-mask-words>[0] .<0:1>= unused .<2> = processor down message .<3> = processor up message .<4> = unused .<5> = process deletion message if D-series format; STOP message if C-series format .<6> = unused if D-series format; ABEND message if C-series format .<7> = unused .<8> = unused if D-series format; MONITORNET message if C-series format .<9> = job creation .<10> = SETTIME message .<11> = power on message .<12> = NEWPROCESS NOWAIT message .<13:15>=unused			

Table 14-3. SET^FILE Operations That Set Addresses (page 3 of 5)

operation	**Description of Operation Requested**	*setaddr-value or new-value*	*old-value*	**State of File**
SET^ SYSTEMMESSAGES MANY (*continued: word 1*)	<sys-msg-mask-words>[1] .<0:3>= unused .<4> = BREAK message .<5> = unused .<6> = time signal message (NonStop II systems only) .<7> = memory lock completion message (NonStop II systems only) .<8> = memory lock failure message (NonStop II systems only) .<9:13>= unused .<14> = open message .<15> = close message	@<new-sys-msg-mask-word>	<sys-msg-mask-words>	

Table 14-3. SET^FILE Operations That Set Addresses (page 4 of 5)

operation	**Description of Operation Requested**	*setaddr-value or new-value*	*old-value*	**State of File**
SET^ SYSTEMMESSAGES MANY (*continued: word 2*)	<sys-msg-mask-words>[2] .<0> = CONTROL message .<1> = SETMODE message .<2> = RESETSYNC message .<3> = CONTROLBUF message .<4:7>= unused .<8> = device-type inquiry if D-series format; unused if C-series format .<9:15>= unused			

Table 14-3. SET^FILE Operations That Set Addresses (page 5 of 5)

operation	Description of Operation Requested	*setaddr-value or new-value*	*old-value*	State of File
SET^ SYSTEMMESSAGES MANY (*continued: word 3*)	<sys-msg-mask-words>[3] .<0> = nowait PROCESS_CREATE_ completion .<1> = subordinate name inquiry .<2> = nowait get info by name completion .<3> = nowait FILENAME_FINDNEXT_ completion .<4> = loss of communication with node .<5> = establishment of communication with node .<6> = remote processor down .<7> = remote processor up .<8:15>= unused			

Example

For pTAL callers:

```
CALL SET^FILE ( IN^FILE , ASSIGN^FILENAME ,,, INFILE^ADDR );
```

For other callers:

```
CALL SET^FILE ( IN^FILE , ASSIGN^FILENAME , @IN^FILENAME );
```

Related Programming Manual

For programming information about the SET^FILE procedure, see the *Guardian Programmer's Guide*.

SETJMP_ Procedure

Summary

The SETJMP_ procedure saves process context in a jump buffer. This context is used when a nonlocal goto is performed by a corresponding call to the LONGJMP_ procedure.

Syntax for C Programmers

```
#include <setjmp.h>

jmp_buf env;

int setjmp ( jmp_buf env );
```

Syntax for TAL Programmers

```
?SOURCE $SYSTEM.ZGUARD.HSETJMP

retval := SETJMP_ ( env );                    ! o
```

Parameters

retval returned value

INT(32)

indicates the outcome of the call:

0D indicates that the SETJMP_ procedure was called directly.

<>0D indicates that the SETJMP_ procedure is returning as a result of a call to the LONGJMP_ procedure. The returned value is specified by LONGJMP_.

env output

INT .EXT:ref:(JMP_BUF_TEMPLATE)

specifies the address of a previously allocated jump buffer in which the process context of the caller is returned. The jump buffer is allocated using the JMP_BUF_DEF DEFINE.

Considerations

- SETJMP_ is the TAL or pTAL procedure name for the C `setjmp()` function. The C `setjmp()` function complies with the POSIX.1 standard.
- Calling SETJMP_ is the functional equivalent of calling the SIGSETJMP_ procedure with *mask* = 0D.
- You can allocate the jump buffer for SETJMP_ using the JMP_BUF_DEF DEFINE as follows:

  ```
  JMP_BUF_DEF ( env );
  ```

  where *env* is a valid variable name.

  Alternatively, you can allocate the buffer by declaring a structure if type JMP_BUF_TEMPLATE.

  In either case, the buffer must be accessible to both the SETJMP_ procedure call and the associated LONGJMP_ procedure call.
- The jump buffer saved by the SETJMP_ procedure is normally used by a call to the LONGJMP_ procedure. The jump buffer can be used by a call to the SIGLONGJMP_ procedure only if the signal mask is not required.
- The buffer pointer is assumed to be valid. An invalid address passed to SETJMP_ will cause unpredictable results and could cause the system to deliver a nondeferrable signal to the process.
- Do not change the contents of the jump buffer. The results of a corresponding LONGJMP_ procedure call are undefined if the contents of the jump buffer are changed.

Example

```
jmp_buf env;

JMP_BUF_DEF_ ( env );
retval := SETJMP_ ( env );
```

Related Programming Manual

For programming information about the SETJMP_ procedure, see the *Guardian Programmer's Guide*.

SETLOOPTIMER Procedure

Summary

A call to the SETLOOPTIMER procedure sets the caller's "process loop-timer" value. A positive loop-timer value enables process loop timing by the operating system and specifies a limit on the total amount of processor time the calling process is allowed. If loop timing is enabled, the operating system decrements the loop-timer value as the process executes (that is, is in the active state). If the loop timer is decremented to 0 (indicating that the time limit is reached), then the timer expires. For a Guardian TNS process, a "process loop-timer timeout" trap occurs (trap number 4). For an OSS process or native process, a `SIGTIMEOUT` signal is generated. Loop timing is disabled by specifying a loop-timer value of 0.

Syntax for C Programmers

```
#include <cextdecs(SETLOOPTIMER)>

_cc_status SETLOOPTIMER ( short new-time-limit
                        ,[ short _near *old-time-limit ] );
```

- The function value returned by SETLOOPTIMER, which indicates the condition code, can be interpreted by the `_status_lt()`, `_status_eq()`, or `_status_gt()` function (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL SETLOOPTIMER ( new-time-limit                    ! i
                  ,[ old-time-limit ] );              ! o
```

Parameters

new-time-limit input

INT:value

specifies the new time-limit value, in 0.01-second units, to be set into the process's loop timer. *new-time-limit* must be a positive value.

If the value of *new-time-limit* is 0, process loop timing is disabled.

old-time-limit output

INT:ref:1

returns the current setting of the process's loop timer (in 0.01-second units).

Condition Code Settings

< (CCL) indicates that the *new-time-limit* parameter is omitted or is specified as a negative value. The state of process loop timing and the setting of the process's loop timer are unchanged.

= (CCE) indicates that the *new-time-limit* value is set into the process's loop timer and that loop timing is enabled.

> (CCG) is not returned from SETLOOPTIMER.

Considerations

- Process processor time

  Using SETLOOPTIMER to measure process processor time is not recommended. Use the MYPROCESSTIME procedure for this purpose.

- Timed asynchronous interrupts

  SETLOOPTIMER is not practical for generating timed asynchronous interrupts for most users.

- Process loop timeout in system code

  If a process loop-timer expires in protected code, the trap (for a Guardian TNS process) or signal (for an OSS process or native process) is delayed until control enters unprotected code.

- Detection of process looping

  To detect whether it is looping, a process can call SETLOOPTIMER (resetting the time limit) at a given point each time through its main execution loop. If the process fails to finish executing its main loop, SETLOOPTIMER is not called and the time limit is not reset. Consequently, the time limit is reached, and a trap or signal occurs. (When the trap handler or signal handler completes execution, the process resumes its normal instruction path.)

For example, a process's main execution loop can be written as follows:

```
 -->
|     |
|  start:
|    CALL SETLOOPTIMER ( 1000 );
|    IF < THEN ... ;
|      |  .
|      |  . ! enable loop timing.
|      |  . ! Time-limit value is 10 seconds.
|      |  .
|    CALL WRITEREAD ( termfnum ,... );
|      |  .
|      |  . ! process executes when terminal input is made.
|      |  . ! Loop-timer value is not decremented while
|      |  . ! process is suspended waiting for I/O.
|      |
|    terminal input is processed.
 -----
```

OSS Considerations

When the process loop-timer expires for an OSS process, a `SIGTIMEOUT` signal is generated.

Example

```
CALL SETLOOPTIMER ( NEW^TIME );
```

SETMODE Procedure

Summary

The SETMODE procedure is used to set device-dependent functions.

A call to the SETMODE procedure is rejected with an error indication if incomplete nowait operations are pending on the specified file.

Syntax for C Programmers

```
#include <cextdecs(SETMODE)>

_cc_status SETMODE ( short filenum
                    ,short function
                    ,[ short param1
                    ,[ short param2
                    ,[ short _near *last-params );
```

- The function value returned by SETMODE, which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL SETMODE ( filenum                    ! i
              ,function                   ! i
              ,[ param1 ]                 ! i
              ,[ param2 ]                 ! i
              ,[ last-params ] );         ! o
```

Parameters

filenum input

INT:value

is a number of an open file that identifies the file to receive the SETMODE *function*.

function input

INT:value

is one of the device-dependent functions listed in Table 14-4 on page 14-63.

param1 input

INT:value

is one of the parameters listed in Table 14-4 on page 14-63. If omitted, for a disk file the present value is retained. For SETMODEs on other devices, this value depends on the device and the value supplied in the *function* parameter.

param2 input

INT:value

is one of the parameters listed in Table 14-4 on page 14-63. If omitted, the present value is retained for disk files; for SETMODEs on other devices, this value depends on the device and the value supplied in the *function* parameter.

last-params output

INT:ref:2

returns the previous settings of *param1* and *param2* associated with the current *function*. The format is:

last-params[0] = old *param1*
last-params[1] = old *param2* (if applicable)

Condition Code Settings

< (CCL) indicates that an error occurred (call FILE_GETINFO_ or FILEINFO).

= (CCE) indicates that the SETMODE is successful.

> (CCG) indicates that the SETMODE function is not allowed for this device type.

SETMODE Functions

Table 14-4 on page 14-63 lists the SETMODE functions that can be used with the I/O devices discussed in this manual for NonStop servers.

Table 14-4. SETMODE Functions (page 1 of 37)

function	**Parameters and Effect**
1	Disk: Set file security *param1* <0> = 1 for program files only, sets accessor's ID to program file's ID when program file is run (PROGID option). <1> = 1 sets CLEARONPURGE option on. This means all data in the file is physically erased from the disk (set to zeros) when the file is purged. If this option is not on, the disk space is only logically deallocated on a purge; the data is not destroyed, and another user might be able to examine the "purged" data when the space is reallocated to another file. <4:6> = ID allowed for reading <7:9> = ID allowed for writing <10:12> = ID allowed for execution <13:15> = ID allowed for purging For each of the fields from <4:6> through <13:15>, the value can be any one of these: 0 = member local ID 1 = member of owner's group (local) 2 = owner (local) 4 = any network user (local or remote) 5 = member of owner's community 6 = local or remote user having same user ID as owner 7 = local super ID only For an explanation of local and remote users, communities, and so forth, see the *Guardian Programmer's Guide* or the *File Utility Program (FUP) Reference Manual* . *param2* is not used with function 1. If this SETMODE function is used on a file that is protected by a Safeguard disk-file authorization record, and if the user is not logged on with the super ID on the system where the file is located, error 199 is returned. This function operates only on Guardian objects. If an OSS file is specified, file-system error 2 occurs.

[1] Function 265 is supported only on systems running H-series RVUs.

Table 14-4. SETMODE Functions (page 2 of 37)

function	**Parameters and Effect**
2	Disk: Set file owner *param1*.<0:7> = group ID .<8:15> = member ID *param2* is not used with function 2. If this SETMODE function is used on a file that is protected by a Safeguard disk-file authorization record, and if the user is not logged on with the super ID on the system where the file is located, error 199 is returned.
2, continued	This function operates only on Guardian objects. If an OSS file is specified, file-system error 2 occurs.
3	Disk: Set write verification *param1*.<15> = 0 verified writes off (default). = 1 verified writes on. *param2* is used with DP2 disk files only. *param2* = 0 change the open option setting of the verify writes option (default). = 1 change the file label default value of the verify writes option.

[1] Function 265 is supported only on systems running H-series RVUs.

Table 14-4. SETMODE Functions (page 3 of 37)

function	**Parameters and Effect**
4	Disk: Set lock mode. Note that this operation is not supported for queue files.
	param1 = 0 normal mode (default): request is suspended when a read or lock is attempted and an established record lock or file lock is encountered.
	= 1 reject mode: request is rejected with file-system error 73 when a read or lock is attempted and an established record lock or file lock is encountered. No data is returned for the rejected request.
	= 2 read-through/normal mode: READ or READUPDATE ignores record locks and file locks; encountering a lock does not delay or prevent reading of a record. The locking response of LOCKFILE, LOCKREC, READLOCK, and READUPDATELOCK is the same as in normal mode (mode 0).
	= 3 read-through/reject mode: READ or READUPDATE ignores record locks and file locks; encountering a lock does not delay or prevent reading of a record. The locking response of LOCKFILE, LOCKREC, READLOCK, and READUPDATELOCK is the same as in reject mode (mode 1).
	= 6 read-warn/normal mode: READ or READUPDATE returns data without regard to record and file locks; however, encountering a lock causes warning code 9 to be returned with the data. The locking response of LOCKFILE, LOCKREC, READLOCK, and READUPDATELOCK is the same as in normal mode (mode 0).
	param1 = 7 read-warn/reject mode: READ or READUPDATE returns data without regard to record and file locks; however, encountering a lock causes warning code 9 to be returned with the data. The locking response of LOCKFILE, LOCKREC, READLOCK, and READUPDATELOCK is the same as in reject mode (mode 1).
	param2 must be 0, if supplied.

[1] Function 265 is supported only on systems running H-series RVUs.

Table 14-4. SETMODE Functions (page 4 of 37)

function	**Parameters and Effect**
5	Line printer: Set system automatic perforation skip mode (assumes standard VFU function in channel 2) *param1*.<15> = 0 off, 66 lines per page = 1 on, 60 lines per page (default) For the 5530 line printer: *param1* = 0 disable automatic perforation skip. = 1 enable automatic perforation skip (default). *param2* is not used with function 5.

6	Line printer or terminal: Set system spacing control
	param1.<15> = 0 no space
	= 1 single space (default setting)
	If *param1*<15> is 0, then
	param2 = 0 adds CR
	param2 = 1 adds nothing (invalid for device subtypes 1, 3, 4, 5, and 6; disables DEV COMPRESS attribute in spooler)
	If *param1*<15> is 1, then
	param2 = 0 adds CR/LF
	param2 = 1 is invalid (might return error)

7	Terminal: Set system auto line feed after receipt of carriage return line termination (default mode is configured) *param1*.<15> = 0 LFTERM line feed from terminal or network (default) = 1 LFSYS system provides line feed after line termination by carriage return *param2* sets the number of retries of I/O operations.
8	Terminal: Set system transfer mode (default mode is configured) *param1*.<15> = 0 conversational mode = 1 page mode *param2* sets the number of retries of I/O operations Note: *param2* is used only with 6530 terminals.

[1] Function 265 is supported only on systems running H-series RVUs.

Table 14-4. SETMODE Functions (page 5 of 37)

function	**Parameters and Effect**
9	Terminal: Set interrupt characters *param1*.<0:7> = character 1 .<8:15> = character 2 *param2*.<0:7> = character 3 .<8:15> = character 4 (Default for conversational mode is backspace, line cancel, end of file, and line termination. Default for page mode is page termination.) See discussion of "interrupt characters" in the *Guardian Programmer's Guide*.
10	Terminal: Set parity checking by system (default is configured) *param1*.<15> = 0 no checking = 1 checking *param2* is not used with function 10.
11	Terminal: Set break ownership *param1* = 0 means BREAK disabled (default setting). = any positive value means enable BREAK. = an internally defined negative value, previously returned in *last-params*, means return BREAK to previous owner. Terminal access mode after BREAK is typed: *param2* = 0 normal mode (any type file access is permitted) = 1 BREAK mode (only BREAK-type file access is permitted) See Section 10, "Communicating With Terminals," and Section 24, "Writing a Terminal Simulator," in the *Guardian Programmer's Guide*.
12	Terminal: Set terminal access mode *param1*.<15> = 0 normal mode (any type file access is permitted) = 1 BREAK mode (only BREAK-type file access is permitted) File access type: *param2*.<15> = 0 normal access to terminal = 1 BREAK access to terminal See the discussion of "Communicating With Terminals" in the *Guardian Programmer's Guide*.

[1] Function 265 is supported only on systems running H-series RVUs.

Table 14-4. SETMODE Functions (page 6 of 37)

function	**Parameters and Effect**
13	Terminal: Set system read termination on ETX character (default is configured) *param1* = 0 no termination on ETX = 1 termination on first character after ETX = 3 termination on second character after ETX *param2* is used, with ATP6100 only, to specify the value of the ETX character. No changes occur to the read termination on ETX as specified by *param1* if you do not specify an ETX character or if you set *param2* to 0 (setting the ETX character to 0 is not allowed).
14	Terminal: Set system read terminal on interrupt characters (default is configured) *param1*.<15> = 0 no termination on interrupt characters (that is, transparency mode) = 1 termination on any interrupt character *param2* is not used with function 14.
SETMODEs 15, 16, 17, 18 and 19 are described in the *EnvoyACP/XF Reference Manual*.	
15	Terminal: Set retry parameters
16	Terminal: Set line parameters
17	Terminal: Set maximum frame size and secondary addresses
18	Terminal: Set statistics threshold, flag fill, and window size
19	Terminal: Set translation parameters and set extended control field size
20	Terminal: Set system echo mode (default is configured) *param1*.<15> = 0 system does not echo characters as read. = 1 system echoes characters as read. *param2* is not used with function 20.

[1] Function 265 is supported only on systems running H-series RVUs.

Table 14-4. SETMODE Functions (page 7 of 37)

function	**Parameters and Effect**
22	Line printer (subtype 3, 4, 6, and 32) or terminal: Set baud rate
	param1 = 0 baud rate = 50
	1 baud rate = 75
	2 baud rate = 110
	3 baud rate = 134.5
	4 baud rate = 150
	5 baud rate = 300
	6 baud rate = 600
	7 baud rate = 1200
	8 baud rate = 1800
	9 baud rate = 2000
	10 baud rate = 2400
	11 baud rate = 3600
	12 baud rate = 4800
	13 baud rate = 7200
	14 baud rate = 9600
	15 baud rate = 19200
	16 baud rate = 200
	17 baud rate = 38400 (5577 printer only)
	param2 is not used with function 22 except when specifying split baud rates with the LIU-4 controller (see below).
	Note:
	The 5520 line printer supports only the 110, 150, 300, 600, 1200, 2400, 4800, and 9600 baud rates.
	The 5530 line printer supports only the 75,150, 300, 600, 1200, 2400, 4800, and 9600 baud rates.
	If no baud rate is specified at configuration time, then 9600 baud is used. The default rate is what was specified at configuration time.
	The asynchronous controller supports only the 150, 300, 600, 1200, and 1800 baud rates.
	An ATP6100 line configured on an LIU-4 controller allows an application to set different transmission (TX) and receiving (RX) baud rates with function 22. You have the option of setting split rates by specifying the TX rate in *param1* and the RX rate in *param2*. You can set nonsplit rates by the normal method (that is, by specifying values for *param1* as listed at the beginning of the description of function 22. The LIU-4 does not support 3600 or 38400 baud rates.)

Table 14-4. SETMODE Functions (page 8 of 37)

function	**Parameters and Effect**
22, continued	You can specify split baud rates with the LIU-4 controller as follows. Note that the values for *param1* all have bit 0 set to 1:
	param1 = 128 TX baud rate = 50
	129 TX baud rate = 75
	130 TX baud rate = 110
	131 TX baud rate = 134.5
	132 TX baud rate = 150
	133 TX baud rate = 300
	134 TX baud rate = 600
	135 TX baud rate = 1200
	136 TX baud rate = 1800
	137 TX baud rate = 2000
	138 TX baud rate = 2400
	140 TX baud rate = 4800
	141 TX baud rate = 7200
	142 TX baud rate = 9600
	143 TX baud rate = 19200
	144 TX baud rate = 200
	param2 = 0 RX baud rate = 50
	1 RX baud rate = 75
	2 RX baud rate = 110
	3 RX baud rate = 134.5
	4 RX baud rate = 150
	5 RX baud rate = 300
	6 RX baud rate = 600
	7 RX baud rate = 1200
	8 RX baud rate = 1800
	9 RX baud rate = 2000
	10 RX baud rate = 2400
	12 RX baud rate = 4800
	13 RX baud rate = 7200
	14 RX baud rate = 9600
	15 RX baud rate = 19200
	16 RX baud rate = 200

Table 14-4. SETMODE Functions (page 9 of 37)

function	**Parameters and Effect**
22, continued	If you specify split baud rates with the LIU-4 controller, the *last-params* parameter returns these values: *last-params*[0].<0:7>*param1* value (TX) .<8:15>*param2* value (RX) *last-params*[1] undefined
23	Terminal: Set character size *param1* = 0 character size = 5 bits 1 character size = 6 bits 2 character size = 7 bits 3 character size = 8 bits *param2* is not used with function 23.
24	Terminal: Set parity generation by system *param1* = 0 parity = odd 1 parity = even 2 parity = none *param2* is not used with function 24.
25	Line printer (subtype 3): Set form length *param1* = length of form in lines *param2* is not used with function 25.
26	Line printer (subtype 3): Set or clear vertical tabs *param1* = 0 is (line#-1) of where tab is to be set. = -1 clear all tabs (except line 1). Note: A vertical tab stop always exists at line 1 (top of form). *param2* is not used with function 26.
27	Line printer or terminal: Set system spacing mode *param1*.<15> = 0 postspace (default setting) = 1 prespace *param2* is not used with function 27.

[1] Function 265 is supported only on systems running H-series RVUs.

Table 14-4. SETMODE Functions (page 10 of 37)

function	**Parameters and Effect**
28	Line printer or terminal: Reset to configured values *param1* = 0 (default) resets printer to its configured values = 1 resets only soft parameters = 2 resets only hard parameters *param2* is not used with function 28. For the 5530 line printer, SETMODE 28 resets all the SETMODE parameters back to their configuration values and also reinitializes the printer. Note: SETMODE 29 (set auto answer or control answer mode) is the only SETMODE function not affected by a SETMODE 28.
29	Line printer (subtype 3, 4, 6, or 32): Set automatic answer mode or control answer mode. *param1*.<15> = 0 CTRLANSWER = 1 AUTOANSWER (default) The default mode is what was specified at configuration time; if no mode is specified at SYSGEN, then AUTOANSWER is used. *param2* is not used with function 29. Note: SETMODE function 29 remains in effect even after the file is closed. SETMODE 29 is the only SETMODE function not affected by a SETMODE 28. SETMODE 29 is not reset at the beginning of each new job. Therefore, you should always issue a SETMODE 29 at the beginning of your job to ensure that you are in the desired mode (rather than in the mode left by the previous job).

[1] Function 265 is supported only on systems running H-series RVUs.

Table 14-4. SETMODE Functions (page 11 of 37)

function	**Parameters and Effect**
30	Allow nowait I/O operations to finish in any order
	param1 = 0 complete operations in the order they were originally requested (default).
	= 1 complete operations in any order, except that if more than one operation is ready to finish at the time of the AWAITIO[X] call, then complete them in the order they were requested.
	= 3 complete operations in any order (that is, in the order chosen by the system).
	param2 is not used with function 30 and should be zero if supplied.
31	Set packet mode
	param1.<0> = 0 ignore *param2*.
	= 1 *param2* specifies leading packet size.
	param2 = 0 use default packet size for transmission (default).
	> 0 is size of first outgoing packet in each WRITE or WRITEREAD request. It must be smaller than the configured packet size.
32	Set X.25 call setup parameters
	param1.<0> = 0 do not accept charge.
	= 1 accept charge.
	.<1> = 0 do not request charge.
	= 1 request charge.
	.<2> = 0 is normal outgoing call.
	= 1 is priority outgoing call.
	.<8:15> = port number (0-99)
	To determine the actual value for port number, see specifications on your own network.
33	Seven-track tape drive: Set conversion mode
	param1 = 0 ASCIIBCD (even parity) (default)
	= 1 BINARY3TO4 (odd parity)
	= 2 BINARY2TO3 (odd parity)
	= 3 BINARY1TO1 (odd parity)

[1] Function 265 is supported only on systems running H-series RVUs.

Table 14-4. SETMODE Functions (page 12 of 37)

function	**Parameters and Effect**
36	Allow requests to be queued on $RECEIVE based on process priority *param1*.<15> = 0 use first-in-first-out (FIFO) ordering (default). = 1 use process priority ordering. *param2* is not used with function 36.
37	Line printer (subtype 1, 4, 5, or 6): Get device status *param1* is not used with function 37. *param2* is not used with function 37. *last-params* = status of device. Status values are: *last-params* for printer (subtype 1 or 5) (only *last-params*[0] is used) .<5> = DOV, data overrun 0 = no overrun 1 = overrun occurred .<7> = CLO, connector loop open 0 = not open 1 = open (device unplugged) .<8> = CID, cable identification 0 = old cable 1 = new cable .<10> = PMO, paper motion 0 = not moving 1 = paper moving .<11> = BOF, bottom of form 0 = not at bottom 1 = at bottom .<12> = TOF, top of form 0 = not at top 1 = at top .<13> = DPE, device parity error 0 = parity OK 1 = parity error .<14> = NOL, not on line 0 = on line 1 = not on line .<15> = NRY, not ready 0 = ready 1 = not ready All other bits are undefined. Note: Ownership, Interrupt Pending, Controller Busy, and Channel Parity errors are not returned in *last-params*; your application program "sees" them as normal file errors. Also, CID must be checked when PMO, BOF, and TOF are tested, because the old cable version does not return any of these states.

[1] Function 265 is supported only on systems running H-series RVUs.

Table 14-4. SETMODE Functions (page 13 of 37)

function	**Parameters and Effect**	
37, continued	*last-params* for printer (subtype 4)	
	last-params [0] = primary status returned from printer:	
	.<9:11> = full status field	0 = partial status
		1 = full status
		2 = full status / VFU fault
		3 = reserved for future use
		4 = full status / data parity error
		5 = full status / buffer overflow
		6 = full status / bail open
		7 = full status / auxiliary status available
	.<12> = buffer full	0 = not full
		1 = full
	.<13> = paper out	0 = OK
		1 = paper out
	.<14> = device power on	0 = OK
		1 = POWER ON error
	.<15> = device not ready	0 = ready
		1 = not ready
	All other bits are undefined.	
	last-params[1] =	auxiliary status word if *last-params*[0].<9:11> = 7; auxiliary status word is as follows:
	.<9:13>=	auxiliary status0= no errors this field
		1= no shuttle motion
		2= character generator absent
		3= VFU channel error
		4-31= reserved for future use
	.<14:15> = always 3	
	All other bits are undefined.	
	last-params for printer (subtype 6)	
	last-params [0] contains the primary status. [1] contains the auxiliary status.	

[1] Function 265 is supported only on systems running H-series RVUs.

Table 14-4. SETMODE Functions (page 14 of 37)

function	**Parameters and Effect**
37, continued	Primary status bits are: [0].<0:8> = 0 undefined .<9> = 1 reserved .<10:12> = 0 no faults = 1 printer idle = 2 paper out = 3 end of ribbon = 4 data parity error = 5 buffer overflow = 6 cover open = 7 auxiliary status available .<13> = 0 buffer not full = 1 buffer full .<14> = 0 OK = 1 device power on error .<15> = 0 OK = 1 device not ready If primary status *last-params*[0].<10:12> = 7, auxiliary status word is: [1].<0:7> = undefined .<8:11> = fault display status (most significant hex digit) .<12:15> = fault display status (least significant hex digit) Fault display status summary:

Operator Display	Aux Status .<8:11>	Aux Status .<12:15>	Problem Description
None	0	0	No faults
E01	0	1	Paper out
E03	0	3	Cover open
E06	0	6	End of ribbon
E07	0	7	Break
E11	1	1	Parity error
E12	1	2	Unprintable character
E22	2	2	Carrier loss
E23	2	3	Buffer overflow
E30	3	0	Printwheel motor fault
E31	3	1	Carriage fault
E32	3	2	Software fault
E34	3	4	Hardware fault

[1] Function 265 is supported only on systems running H-series RVUs.

Table 14-4. SETMODE Functions (page 15 of 37)

function	**Parameters and Effect**
38	Terminal: Set special line-termination mode and character
	param1 = 0 sets special line-termination mode. *param2* is the new line-termination character. The line-termination character is not counted in the length of a read. No system-supplied carriage return or line feed is issued at the end of a read (see note on cursor movement below).
	= 1 sets special line-termination mode. *param2* is the new line-termination interrupt character. The line-termination character is counted in the length of a read. No system-supplied carriage return or line feed is issued at the end of a read (see note on cursor movement below).
38, continued	= 2 resets special line-termination mode. The line-termination interrupt character is restored to its configured value. *param2* must be present but is not used.
	param2 = the new line-termination interrupt character (passed in bits <8:15>) if *param1* = 0 or 1.
	last-params if present, returns the current mode in *last-params*[0] and the current line-termination interrupt character in *last-params* [1].
	Note:
	Although the cursor typically will not move when 0 or 1 is specified for *param1*, these options do not turn off ECHO. Therefore, if the termination character is one which would normally cause cursor movement (such as a LF or CR) and ECHO is enabled, cursor movement will occur.
SETMODEs 40-49 are reserved for the Exchange products. For details, see the Exchange reference manuals.	
50	Enable/disable 3270 COPY
	param1 = 0 suppress COPY
	= 1 allow COPY
51	Get/set 3270 status
	param1 status flags mask to set
	param2 is not used with function 51
	For the flags mask information, see the *Device-Specific Access Methods - AM3270/TR3271* manual.

[1] Function 265 is supported only on systems running H-series RVUs.

Table 14-4. SETMODE Functions (page 16 of 37)

function	**Parameters and Effect**
52	Tape drive: Set short write mode
	param1 = 0 allow writes shorter than 24 bytes; a record shorter than 24 bytes is padded with zeros to a length of 24 bytes (default).
	= 1 disallow writes shorter than 24 bytes.
	= 2 allow writes shorter than 24 bytes; no padding is done on records shorter than 24 bytes.
	param2 is not used with function 52.
	Note:
	When short writes are disallowed, an attempt to WRITE or WRITEUPDATE a record that is shorter than 24 bytes causes error 21 (bad count) to be issued.
53	Enable/disable receipt of status
	param1 = 0 disable status receive
	= 1 enable status receive
	param2 = the response ID
54	Return control unit and device assigned to subdevice
	param1 is not used with function 54.
	param2 is not used with function 54.
	last-params [0] .<0:7>= 0
	.<8:15>= subdevice number known by AM3270
	[1] .<0:7>= standard 3270 control-unit address
	.<8:15>= standard 3270 device address
57	Disk: Set serial writes option
	param1 = 0 system automatically selects serial or parallel writes (default).
	= 1 use serial writes unconditionally.
	param2 is used with DP2 disk files only.
	param2 = 0 The setting of *Param1* will affect this open only. Other opens of the file are not affected
	= 1 The setting of *Param1* will affect this open and all future opens of the file. Other existing opens are not affected.
59	Return count of bytes read
	param1 = count of actual bytes read
	param2 = 0

[1] Function 265 is supported only on systems running H-series RVUs.

Table 14-4. SETMODE Functions (page 17 of 37)

function	**Parameters and Effect**
66	Tape drive: Set density *param1* = 0 800 bpi (NRZI) = 1 1600 bpi (PE) = 2 6250 bpi (GCR) = 3 as indicated by switches on tape drive = 8 38000 bpi *param2* is not used with function 66.
67	AUTODCONNECT for full-duplex modems: Monitor carrier detect or data set ready *param1* = 0 disable AUTODCONNECT (default setting). = 1 enable AUTODCONNECT. *param2* is not used with function 67.
68	Line printer (subtype 4): Set horizontal pitch *param1* = 0 normal print (default) = 1 condensed print = 2 expanded print *param2* is not used with function 68.
71	Set transmission priority. Transmission authority is used by an Expand process to determine the ordering of messages for transmission on an Expand path. This operation indirectly invokes the network utility process, $ZNUP, on a remote system to get information before the request can be serviced. *param1* is not used with function 71. *param2* .<0:7> = 0 (reserved) .<8:15> = transmission priority The transmission priority value can range from 0 through 255. A value of 0 (the default) causes the processor priority of the process to be used as the transmission authority. 1 is the lowest priority; 255 is the highest priority. Once a path is selected, the Expand process processes the highest-priority message first.

[1] Function 265 is supported only on systems running H-series RVUs.

Table 14-4. SETMODE Functions (page 18 of 37)

function	**Parameters and Effect**
72	Force system buffering for nowait files *param1* = 0 allow the system to make transfers directly from user buffers. This is applicable for systems running G-series RVUs. For systems running H-series RVUs, if the USERIOBUFFER_ALLOW_ procedure has been called or if the user_buffers flag is set to ON in the object file, user buffers are allowed. Otherwise, only PFS buffers are used for the file. = 1 force use of intermediate buffer in PFS. (Some nowaited write operations on the file with alternate keys will be forced to wait. Nowait write operations always require the data to remain unchanged until the information is completed.) The default value for files opened by FILE_OPEN_ is 0. For files opened by OPEN, the default value is 1. = 2 user buffers are allowed for this file after this call regardless of the previous setting. User buffers being allowed does not guarantee that the user buffers will be used; the system is still free to select the most efficient buffers to use. In practice, I/O less than 4096 bytes will use PFS buffers. *param2* is not used and should be zero if supplied. *last-params* reflects the current effective SETMODE 72 value for this file, which is either 1 or 2, and not 0. This is applicable for systems running J06.05 and later J-series RVUs and H06.16 and later H-series RVUs. For example, if the USERIOBUFFER_ALLOW_ procedure is called before the file is opened, the *last-params* will return 2.

[1] Function 265 is supported only on systems running H-series RVUs.

Table 14-4. SETMODE Functions (page 19 of 37)

function	**Parameters and Effect**
80	($RECEIVE): Set system message modes *param1* .<0:12> should be zero. .<13> = 0 disable reception of -38 messages (default). = 1 enable reception of cancellation (-38) messages. .<14> = 0 disallow reception of SETPARAM (-37) messages, returning error 2 to processes attempting SETPARAM calls (default). = 1 allow reception of SETPARAM (-37) messages. .<15> = 0 disallow return of <last-param> values for SETMODE (-33) messages, returning error 2 to processes attempting to obtain them (default). = 1 allow <last-param> values to be returned for SETMODE (-33) messages. The extended form of the -33 message will be delivered under this mode. *param2* is not used and should be zero if supplied.
90	Disk: Set buffered option defaults same as CREATE *param1* = 0 buffered = 1 write-through *param2* is used with DP2 disk files only. *param2* = 0 change the open option setting of the buffered option (default). = 1 change the file label default value of the buffered option.
90, continued	This function operates only on Guardian objects. If an OSS file is specified, file-system error 2 occurs.

[1] Function 265 is supported only on systems running H-series RVUs.

Table 14-4. SETMODE Functions (page 20 of 37)

function	**Parameters and Effect**
91	Disk: Set cache and sequential option (function 91 is not applicable for alternate-key files). *param1* = 0 system managed (default). DP2 will detect sequential access; when detected, it will set LRU access to sequential and perform key-sequenced sequential splits. = 1 direct I/O, bypass disk cache = 2 random access, LRU-chain buffer = 3 sequential access, reuse buffer. Directs DP2 to set cache LRU access to sequential and perform key-sequenced sequential splits. *param2* is not used with function 91. Sequential LRU access results in “random” LRU chaining that provides an approximate half-life within the LRU cache chain. Key-sequenced sequential splits attempt to leave the inserted record as the last in the old block, in contrast to a 50/50 split. This helps to ensure a compact key-sequenced structure when multiple sequential records are inserted.
92	Disk: Set maximum number of extents for a nonpartitioned file. (Function 92 is invalid for audit-trail files and for partitioned non-key-sequenced files.) *param1* = new maximum number of extents value. There is no guarantee of success if you specify a value greater than 500. The default is 16 extents. *param2* is not used with function 92. This SETMODE operation returns ERROR 12 (file in use) if: ● the file is a format 2 file, non-keysequenced and non-partitioned ● the new values for maxextents would cause the maximum size of the file to exceed 4 GB ● the maximum size of the file before the SETMODE is less than 4 GB ● the file is currently opened by the process issuing the SETMODE and the open did not specify the "use 64-bit primary keys" election to FILE_OPEN_

[1] Function 265 is supported only on systems running H-series RVUs.

Table 14-4. SETMODE Functions (page 21 of 37)

function	**Parameters and Effect**
93	Disk: Set buffer length for an unstructured file *param1* = new BUFFERSIZE value, must be valid DP2 block size. Valid DP2 block sizes are 512,1024, 2048, 4096 bytes (the default is 4096 bytes). *param2* is not used with function 93. This function operates only on Guardian objects. If an OSS file is specified or if the specified Guardian file is opened by an OSS function, file-system error 2 occurs.
94	Disk: Set audit-checkpoint compression option for an Enscribe file. *param1* = 0 no audit-checkpoint compression (default) = 1 audit-checkpoint compression enabled *param2* = 0 change the open option setting of the audit-checkpoint compression option (default). = 1 change the file label default value of the audit-checkpoint compression option.
95	Disk: Flush dirty cache buffers *param1* is not used with function 95. *param2* is not used with function 95. If *last-params* is specified, SETMODE returns: 0 = broken file flag off after dirty cache blocks were written to disk 1 = broken file flag on, indicating some part of file is bad, possibly due to failed write of a dirty cache block
97	License program to use privileged procedures *param1* = 0 disallow privilege (revoke license) = 1 allow privilege (license) *param2* is not used with function 97. To use this SETMODE function on a Guardian object, the user must be logged on as the super ID on the system where the file is located. To use this SETMODE function on an OSS object, the user must have appropriate privilege; that is, the user must be locally authenticated as the super ID on the system where the target object resides.

[1] Function 265 is supported only on systems running H-series RVUs.

Table 14-4. SETMODE Functions (page 22 of 37)

function	**Parameters and Effect**
99	TAPEPROCESS error recovery method *param1* = 0 record level recovery (default) The record is written to tape as soon as it is received. = 1 file level recovery Data is buffered at the drive until an end-of-file mark is received. The data is then flushed from the drive buffer to the tape media. = 2 volume level recovery Data and end-of-file marks are buffered until the device sees two consecutive end-of-file marks or a tape_synch. The device buffer is then flushed from the drive buffer to the tape media. Once set, the buffering method remains set until changed or until the application issues a FILE_CLOSE_ on the device corresponding to a previous FILE_OPEN_. At that time the buffering is reset to record-level at the device. *param2* is not used with function 99.
SETMODEs 100-109 are reserved for customer use.	
110	Set Shift In/Shift Out (SISO) code extension technique for an individual subdevice Note: SETMODE 110 is supported in AM6520 for CRT protocol when used for 6530 terminals, ITI protocol for 6530 terminals, and PRT protocol for 5520 printers. *param1* = 0 disable SISO (default setting) = 1 enable SISO *param2* is not used with function 110.
112	Session in Between Brackets (BETB) state Places SNAX LU-LU session in BETB state so that either the first speaker or the bidder can bid to open a new bracket. *param1* = 0 disable = 1 enable placing session in the BETB state

[1] Function 265 is supported only on systems running H-series RVUs.

Table 14-4. SETMODE Functions (page 23 of 37)

function	**Parameters and Effect**
113	Set screen size *param1* = the screen width (40, 66, 80, or 132) *param2* = the screen length (25 or 28) *last-params* is an integer reference parameter containing two elements: *last-params* [0] = old *param1* [1] = old *param2*
115	SNA Control Request Notification *param1* = 0 disable = 1 enable control request notification Users of CRT protocol can receive notification that a CINIT or CTERM request has been sent by the SSCP to the PLU.
116	Terminal: Establish extended address field for ADCCP (ABM only) combined stations. See the *EnvoyACP/XF Reference Manual*.
117	Process files: Set TRANSID forwarding *param1* = 0 normal mode (default for process subtypes other than 30 and 31): If a transaction identifier is in effect at the time of a write, read, or writeread on the process file, it is sent with the request so that the receiver will operate under the transaction. = 1 suppress mode (default for process subtypes 30 and 31): A transaction identifier is never associated with a message on the process file, whether or not one is in effect for the sending process. The SETMODE action is local to the program that calls SETMODE; no SETMODE message is sent to the destination process. If this SETMODE function is invoked in a process pair, provision should be made to either call CHECKSETMODE or to reexecute the SETMODE call at the time of a takeover.
119	Tape drive: Set mode *param1* = 0 set start/stop mode = 1 set streaming mode *param2* is not used with function 119. Only the 5120 tape drive supports both options of SETMODE 119. The 5130 tape drive supports this SETMODE function only when *param1* = 0. The rest of the supported tape drives support this SETMODE function only when *param1* = 1.

[1] Function 265 is supported only on systems running H-series RVUs.

Table 14-4. SETMODE Functions (page 24 of 37)

function	**Parameters and Effect**
120	Return end-of-tape (EOT) message when writing labeled tapes *param1* = 0 volume switching is transparent = 1 notify user of volume switch by sending error 150 (EOT). COBOL applications do not receive error 150 (EOT); the COBOL run-time library (RTL) handles this error transparently. *param2* is not used with function 120.
123	Disk: Set the generic lock key length of a key-sequenced file (DP2 only). Note that this operation is not supported for queue files. *param1* = lock key length. The generic lock key length determines the grouping of records that will share a single lock. This value must be between 0 and the key length of the file. If locks are in force at the time of the call, this SETMODE function is rejected with error 73. The key length value applies to all partitions of a file. Alternate keys are not affected. If the lock key length is nonzero and less than the key length of the file, the keys are not affected. If the lock key length is nonzero and less than the key length of the file, generic locking is activated and calls to UNLOCKREC are thus ignored. Generic locking is turned off by giving a lock key length of 0 or equal to the key length of the file (which is equivalent to 0). *param2* is not used and should be zero if supplied. Note that when generic locking is activated, any write is rejected with error 73 if it attempts to insert a record having the same generic lock key as an existing lock owned by another user, whether for audited or nonaudited files.

[1] Function 265 is supported only on systems running H-series RVUs.

Table 14-4. SETMODE Functions (page 25 of 37)

function	**Parameters and Effect**
128	Queue files: Specifing timeout periods. *param1* = the higher order word of the timeout value (in 0.01 second). *param2* = the lower order word of the timeout value (in 0.01 second). The two words are combined to form a 32-bit integer for the timeout value. These values are reserved: -2D = system default period (60 seconds) -1D = infinite timeout period (timeout error is not returned) 0D = no timeout period (error is returned immediately if record cannot be read) All other negative values are invalid. Note that SETMODE function 128 is valid for queue files only. The purpose of the timeout period is to limit the time spent on dequeue operations, especially for audited files. If the read operation is not completed within the timeout period, an error 162 (operation timed out) is returned.

[1] Function 265 is supported only on systems running H-series RVUs.

Table 14-4. SETMODE Functions (page 26 of 37)

function	**Parameters and Effect**
141	DP2 disk file: Enable/disable large transfers *param1* = 1 enable large transfers = 0 disable large transfers (this is the default value when the file is opened.) *param2* is not used with function 141. *last-params*[0] contains the previous setting of the large transfer mode flag. *last-params*[1] if *param1* is 1, this contains the value of the file's broken flag after the cache has been flushed. if *param1* is 0, this contains a 0. SETMODE 141 is valid only for DP2 disk files. When enabled by a SETMODE 141 (or SETMODENOWAIT 141), a read or write operation can transfer up to 56K bytes of data to a DP2 disk file. This setmode is in most cases now unnecessary for unstructured files (see READ or WRITE Considerations). If this SETMODE function is waited and any nowait I/O operations are outstanding at the time SETMODE is called, the SETMODE is not done. The condition code is set to CCL and error 27 is returned. When this SETMODE function is issued with *param1* set to 1, DP2 flushes and removes all blocks for the file from its cache. This ensures that any updates done by the user before the SETMODE are written to disk. The file must have been opened with the unstructured access option specified, even if the file is unstructured. There is no support for alternate-key files or partitions. Because the file is opened for unstructured access, secondary partitions and alternate-key files are not opened.

[1] Function 265 is supported only on systems running H-series RVUs.

Table 14-4. SETMODE Functions (page 27 of 37)

function	**Parameters and Effect**
141, continued	If the file is audited and structured, the file must be opened with an access mode of read-only. The exclusion mode can be shared, protected, or exclusive. If the exclusion mode is shared, updates done by other openers of the file might not be seen by this opener. If the file is audited and unstructured, only read operations can be performed on the file after this SETMODE is issued with *param1* set to 1. Write operations must not be issued until the SETMODE is reissued with *param1* set to 0. If the file is not audited, the file can be opened with any access mode and any exclusion mode. If the exclusion mode is shared, updates done by other openers of the file might not be seen by this opener. To issue write requests after the SETMODE is issued with *param1* set to 1, the exclusion mode must be exclusive. Once the large transfer mode is enabled, only the data transfer operations of READ[X], READUPDATE[X], WRITE[X], and WRITEUPDATE[X] are allowed. No record locks are supported. READ[X] and WRITE[X] use the record address in NEXTREC. READUPDATE[X] and WRITEUPDATE[X] use the record address in CURREC. POSITION can be used to set CURREC and NEXTREC. Relative byte addressing is used for positioning. The operation is done nowait if the file is opened for nowait I/O. The operation must then be completed by a call to AWAITIOX. The operation can be canceled by CANCEL or CANCELREQ. With the large transfer mode enabled, data is read or written directly from the user's buffer. The user's buffer is locked in memory until completion of the operation. The data is not moved to or from the PFS (regardless of the setting of SETMODE 72). If the I/O is done nowait, the user should not modify or examine the data in the buffer until the I/O has finished. This also applies to other processes sharing the segment containing the buffer. If the I/O is done in a nowait manner and the buffer is in the stack, the buffer must be in the high end of the stack. The record address (in NEXTREC or CURREC) must be on a page boundary (a multiple of 2K bytes) or error 550, "invalid position," will be returned. Note that, as usual, when performing successive READ[X]s without intervening POSITIONs, NEXTREC is incremented by the count actually read, and that just before the EOF, the count read will probably not be a multiple of 2K bytes, hence violating the record address constraint for the next READ[X] call. A POSITION before each READX call ensures that the record address meets the page boundary constraint.

[1] Function 265 is supported only on systems running H-series RVUs.

Table 14-4. SETMODE Functions (page 28 of 37)

function	**Parameters and Effect**
141, continued	f -1D positioning is used, the current EOF must be on a page boundary, otherwise the disk process returns an error. If -1D positioning is used or writes are done that change the EOF, some performance gains are lost. When the EOF is changed, DP2 must do extra checkpoints. To avoid the extra checkpoints, issue a CONTROL 2 to set the EOF to a high value before starting a series of large writes. When the writes are complete, another CONTROL 2 can be issued to set EOF to the correct value. The length of the transfer cannot be more than 57344 bytes. The length must be a multiple of 2K bytes. Also, some older Expand connections do not support transfers of more than 30K bytes and return an error if a larger transfer is attempted. Large transfer requests are considered repeatable, so no sync ID is passed to the disk process. If the file is opened with a positive sync depth, normal retries occur after path failures. A positive sync depth should not be used with -1D positioning, because writes to the end of a file are not repeatable. If a path failure occurs, determine whether the data was written before the path failure. If the data was not written, issue a retry. Since DP2's cache is bypassed by these operations and the data is read from or written to the disk file directly, records or blocks modified by other openers might not be seen by these requests. The user might see "dirty" data if access mode and exclusion mode are not set carefully.
142	Select Character Set *param1* = 1 select IBM PC character set. = 0 select default character set. *param2* is not used with function 142. If supplied, it must be 0. For 5512, 5515/5516, and 5573 printers, the I/O process sends these ESC sequences to select the character set: ESC(1OU select IBM PC character set. ESC(1O@ select default character set. Note: These printers do not come equipped with the IBM PC character set; it can be installed later. To use this operation, the printer must have the IBM PC character set installed on the printer. If the printer does not have the IBM PC character set installed, the printer ignores the SETMODE 142.

[1] Function 265 is supported only on systems running H-series RVUs.

Table 14-4. SETMODE Functions (page 29 of 37)

function	**Parameters and Effect**
144	Sets LU character set and double-byte character code *param1* must be omitted for function 144. *param2* must be omitted for function 144. *last-params*[0].<0> = 0 no translation = 1 SNAX does EBCDIC/ASCII translation *last-params*[0].<1:7> = IBM device type 1=IBM-3277 2=not 3277 or 3276 3=IBM-3276 *last-params*[0].<8:15> = value of LU attribute ALLOWEDMIX *last-params*[1].<0:7> = LU character set 0=ASCII (USASCII) 9=EBCDIC (IBM EBCDIC) 14 =KATAKANA EBCDIC *last-params*[1].<8:15> = double-byte character set 0=No DBCS 2=IBMKANJI 3=IBMMIXED 5=JEFKANJI
146	Queue waiters for disk file write. Note that this operation is not supported for queue files. *param1* = 0 disables the effect of *param1* = 1 causes CONTROL 27 requests to be queued and completed one at a time; otherwise all are completed by the first write. *param2* is not used with function 146. SETMODE 146 remains in effect until the file has no more openers. When there are no more openers, the effect of SETMODE 146 is lost and the next open must repeat this SETMODE function.

[1] Function 265 is supported only on systems running H-series RVUs.

Table 14-4. SETMODE Functions (page 30 of 37)

function	**Parameters and Effect**
148	Disk: Wait for insert to locked file. *param1* = 0 normal mode (default). Request is rejected with file-system error 73 when insert is attempted and an established file lock is encountered. = 1 wait mode. Request is suspended when insert is attempted and an established file lock is encountered.
149	Disk: Alternate key insertion locking. Note that this operation is not supported for queue files. *param1* = 0 no automatic locking (default): locking and unlocking during insertion is not automatically performed. = 1 automatic locking: writes of record with alternate keys are locked at the beginning of insertion and unlocked after all associated alternate key record insertions are complete. This prevents interference from programs attempting concurrent update of the same record. *param2* must be zero if supplied. SETMODE 149 is not valid (nor needed) for audited files. For process pairs, the mode needs to be set in the backup by CHECKSETMODE. If the file is a key-sequenced file using the generic locks feature, the lock established by automatic locking might not be released automatically at the end of the write processing.

[1] Function 265 is supported only on systems running H-series RVUs.

Table 14-4. SETMODE Functions (page 31 of 37)

function	**Parameters and Effect**
152	Disk: Reduce the overhead of multiple closes of a file that has multiple openers by flushing the cache on the last close of the file. *param1*.<15> = 1 for nonaudited disk files, causes the cache not to be flushed unless the file is closed by the last opener with either write access or a nonzero sync-depth value. To be effective, SETMODE 152 with *param1*.<15> = 1 should be performed for each open of a file. = 0 causes the cache to be flushed when this opener closes the file, if the file is opened for write access. The default action is to flush the cache only when the last write-access opener closes the file. *param1*.<15> = 1 disables the effect of SETMODE 152 with *param1*.<15> = 0. *param2* is not used with function 152. If it is supplied, it must be 0. SETMODE 152 with *param1*.<15> = 1 postpones the flush of the cache until the last opener with either write access or a nonzero sync-depth value closes the file. For example, an application can control cache flushing by creating a persistent opener and closing the file when more system resources are available. For optimal performance, do not use SETMODE 152 with *param1*.<15> = 1 if the file is closed by all openers at about the same time, because all buffers in the cache are closed serially by the last opener rather than in parallel by each opener.

[1] Function 265 is supported only on systems running H-series RVUs.

Table 14-4. SETMODE Functions (page 32 of 37)

function	**Parameters and Effect**
153	Disk: Variable-length audit compression. (D10 and later RVUs. This function should be avoided if a fallback to a RVU earlier than D10 is possible.) Enable or disable variable-length audit compression for any structured Enscribe file. *param1* = 1 enables variable-length audit compression. = 0 disables variable-length audit compression. *param2* is not used with function 153. If supplied, it must be 0. CAUTION: Since this feature generates undo and redo that cannot be processed correctly by earlier RVUs of the disk process, a fallback to a RVU earlier than D10 can result in problems. If a fallback to a RVU earlier than D10 is possible, this function should be avoided. If this function is used and a fallback becomes necessary, these steps should be taken: 1) TMF should be stopped while the affected volumes are brought up. TMF can subsequently be restarted. 2) Once the earlier version of operating system is running, new TMF online dumps should be acquired for any TMF-protected files on which SETMODE function 153 was used if TMF rollforward protection is required.
158	Disk: Makes the validation of sector and block checksums optional for read opeartions. *param1* = 0 Validate sector and block checksums on subsequent read operations. = 1 Omit sector and block checksum validation on subsequent read operations. *param2* is not used with function 158. If supplied, it must be 0. *last-params*[0] contains the checksum setting. SETMODE 158 does not validate the *param1* argument and does not generate an error message if the argument is not valid. By default, checksums are validated from disk I/O operations. SETMODE function 158 request is ignored for outstanding read operations. The file must be opened with unstrusctured-access option specified, even if the file is unstructured. Function 158 does not support alternate-key files or partitions.

[1] Function 265 is supported only on systems running H-series RVUs.

Table 14-4. SETMODE Functions (page 33 of 37)

function	**Parameters and Effect**
162	Override System Compression Default on 5190 Cartridge Tape *param1* = 1 No data compression = 2 Data compression (IDRC) *param2* is not used with function 162. Users of unlabeled tapes who do not want to use the default compression setting can use SETMODE 162 to override the default setting. BACKUP and FUP do not support this operation. This operation is allowed only at the beginning of tape (BOT). For more information about the 5190 Cartridge Tape Subsystem, see the *5190 Cartridge Tape Subsystem Manual*
163	SNAX:Enhanced CDI mode *param1* = 0 enables normal mode (disables all enhanced CDI support). = 1 enables enhanced CDI mode. = 2 enables special WRITEREAD mode (allows applications to determine the setting of CDI by using the WRITEREAD procedure; using WRITEREAD causes the outbound data buffer to be sent with CDI enabled). *param2* is not used with function 163. A SETMODE 163 call applies only to the opener, not to the device that is opened; another SETMODE 163 call must be made if the file is closed. Enhanced CDI mode is supported only by the SNAX/XF and SNAX/CDF products. For more information about enhanced CDI mode, see the *SNAX/XF Application Programming Manual* and the *SNAX/CDF Application Programming Manual.*

[1] Function 265 is supported only on systems running H-series RVUs.

Table 14-4. SETMODE Functions (page 34 of 37)

function	**Parameters and Effect**
165	SNAX:Exception response (ER) mode *param1* = 0 disables ER mode. The LU sends outbound LU-LU FMD requests in definite response (DR) mode. This is the default value if SCF or SPI has not been used previously to configure ER mode; otherwise, the SCF or SPI configuration value is used. = 1 enables ER mode for applications existing before the introduction of ER mode. The LU sends outbound LU-LU FMD requests in ER mode. However, error 122 is sent on a negative response instead of error 951. = 2 enables ER mode for applications using the ER mode features. The LU sends outbound LU-LU FMD requests in ER mode. Error 951 is sent on a negative response. *param2* is not used with function 165. *last-params*[0] contains the previous value of *param1*. *last-params*[1] contains the number of outbound requests required per device before an ER mode sync point is automatically generated on behalf of the user. ER mode is supported only by the SNAX/XF product. Specifying this function for SNALU causes an error 2 (invalid operation) to be returned. Passing a value other than one of those listed above causes an error 590 (invalid parameter value) to be returned. A SETMODE 165 call applies only to the opener, not to the device that is opened; another SETMODE 165 call must be made if the file is closed. For more information about ER mode, see the *SNAX/XF Application Programming Manual.*
258	Telserv: Transfers data in either half duplex mode or full duplex mode. *param1* .<15> = 0 Normal half duplex mode (default). .<15> = 1 Full duplex mode. *param2* is not used with function 258. In the full duplex mode,write requests are not queued behind read requests; they are processed immediately. Writeread requests are allowed only if there are no pending requests; otherwise, error 160 is returned. **Cause.** For more information, see the *Telserv Guide*.

[1] Function 265 is supported only on systems running H-series RVUs.

Table 14-4. SETMODE Functions (page 35 of 37)

function	**Parameters and Effect**
260	Printer: Enables PostScript printing for the FASTP print process and FASTP-based print processes. *param1* = 2 The FASTP print process sends the PCL command to switch the printer to PostScript mode. At the end of each line, a system-generated carriage return is issued. At the end of the job and before the next job prints, the printer is returned to PCL mode. =1 The FASTP print process sends the PCL command to switch the printer to PostScript mode. At the end of the job and before the next job prints, the printer is returned to PCL mode. (No system-generated carriage return is issued at the end of each line.) = 0 The FASTP print process sends the PCL command to switch the printer back to PCL mode. *param2* is not used with function 260. Function 260 applies only to 5577 printers. When PostScript mode is in effect, SETMODE 142 and CONTROL 1 are ignored and FASTP inhibits the sending of other PCL sequences such as page eject and job offset. For programming information about SETMODE function 260, see the *Guardian Programmer's Guide*.
261	Telserv: Enables/Disables expand tabs (09 hex); applies to echo and output. *param1* = 1 enables TAB to blank expansion. Equivalent startup option is -XTABS (default). = 0 disables conversion. Equivalent startup option is -NOXTABS. *param2* is not used with function 261. For more information about -XTABS and -NOXTABS, see the *Telserv Guide*.
262	Telserv: Enables/Disables the conversion of nonprinting characters. *param1* = 1 enables conversion of nonprinting characters. Equivalent startup option is -CTRLECHO (default). = 0 disables this conversion. Equivalent startup option is -NOCTRLECHO. *param2* is not used with function 262. For more information about -CTRLECHO and -NOCTRLECHO, see the *Telserv Guide*.

[1] Function 265 is supported only on systems running H-series RVUs.

Table 14-4. SETMODE Functions (page 36 of 37)

function	**Parameters and Effect**
263	Telserv: Enables/Disables processing of CTRL-C (03 hex) as break character. *param1* = 1 enables processing of CTRL-C (03 hex) as break character. Equivalent startup option is -BREAKDATA (default). = 0 disables processing of CTRL-C (03 hex) as break character. Equivalent startup option is -NOBREAKDATA. *param2* is not used with function 263. Note that IAC BREAK will still be processed. For more information about -BREAKDATA and -NOBREAKDATA, see the *Telserv Guide.*
264	Telserv: Enables/Disbles processing of CTRL-S and CTRL-Q characters as XON and XOFF. *param1* = 1 enables the processing of CTRL-S and CTRL-Q characters as flow control characters(XON and XOFF respectively). The equivalent startup option is -FLOWCTRL (default). = 0 disables the processing of CTRL-S and CTRL-Q characters as flow control characters. Instead they are interpreted as normal data characters. The equivalent startup option is -NOFLOWCTRL. *param2* is not used with function 264. For more information about -FLOWCTRL and -NOFLOWCTRL, see the *Telserv Guide*.
265[1]	Disk: Set the TRUST flag that controls the direct I/O access permission to user buffers when the process is running. *param1* = 0 disables the TRUST flag. = 1 enables the TRUST flag, allowing private access to the process. = 3 enables the TRUST flag, allowing shared access to the process. *param2* is not used with function 265. To use the SETMODE function on a Guardian file, the user must log on as the super ID on the system. To use the SETMODE function on an OSS file, the user must have the appropriate privilege. That is, the user must be locally authenticated as the super ID on the system.

[1] Function 265 is supported only on systems running H-series RVUs.

Table 14-4. SETMODE Functions (page 37 of 37)

function	**Parameters and Effect**
266	Specify whether to defer the flush of a file's dirty blocks after all the openers have closed the file. The setting of `param1` determines the behavior. If multiple concurrent openers issue SETMODE 266 for the same file, the value of `param1` specified by the most recent request takes effect. *param1* = 0 Possibly flush the specified file's dirty blocks during file close processing. This is the default behavior. = 1 Do not flush the specified file's dirty blocks during file close processing. Using SETMODE 266 with nonzero sync-depth might have unexpected consequences if the file is also opened with sync-depth = 0. If the open with nonzero `sync-depth` is closed while a `sync-depth` = 0 open remains, the checkpointed buffers will be discarded by the Backup Disk Process. A subsequent failure of the primary CPU, disk process, or primary disk paths could result in data being lost for buffers that are not yet written. SETMODE 266 can be done only on structured, nonaudited files. This operation on unstructured or audited files will fail with the error EINVALOP.

[1] Function 265 is supported only on systems running H-series RVUs.

Considerations

- Default SETMODE settings

  The SETMODE settings designated as “default” are the values that apply when a file is opened (not if a particular *function* is omitted when SETMODE is called).

- Waited SETMODE

  The SETMODE procedure is used on a file as a waited operation even if *filenum* has been opened for nowait. Use the SETMODENOWAIT procedure for nowait operations.

- No SETMODEs on Telserv are allowed before doing a CONTROL 11.

Disk File Consideration

- Ownership and security of file

  “Set disk file security” and “set disk file owner” are rejected unless the requester is the owner of the file or the super ID.

Interprocess Communication Considerations

- Nonstandard parameter values

Any value can be specified for the *function*, *param1*, and *param2* parameters. An application-defined protocol should be established for interpreting nonstandard parameter values.

- User-defined SETMODEs

  Use of function code numbers 100 to 109 avoids any potential conflict with SETMODE codes defined by HP.

- Incorrect use of *last-params*

  Error 2 is returned when the *last-params* parameter has been supplied but the target process does not correctly return values for this parameter.

Messages

- Process SETMODE message

  Issuing a SETMODE to a file representing another process causes a system message -33 (process SETMODE) to be sent to that process.

  The identification of the process that called SETMODE can be obtained in a subsequent call to FILE_GETRECEIVEINFO_ (or LASTRECEIVE or RECEIVEINFO). (For a list of all system messages sent to processes, see the *Guardian Procedure Errors and Messages Manual*.)

Examples

```
1.      LITERAL SECURITY = %0222;
            .
            .
            .
        CALL SETMODE ( FNUM , 1 , SECURITY );
```

The LITERAL above sets the file's security to:

```
read    =  any local user
write   =  owner only
execute =  owner only
purge   =  owner only
```

```
2.      LITERAL PROG^SEC = %102202;
            .
            .
            .
        CALL SETMODE ( PFNUM , 1 , PROG^SEC );
```

This LITERAL specifies that the file's owner ID should be used by the calling process as its process access ID when the program file is run. This is done by setting the file's security to:

set process access ID to owner's ID when file is run

read	owner only
write	owner only
execute	any local user
purge	owner only

Related Programming Manuals

For programming information about the SETMODE file-system procedure, see the *Guardian Programmer's Guide* and the data communication manuals.

SETMODENOWAIT Procedure

Summary

The SETMODENOWAIT procedure is used to set device-dependent functions in a nowait manner on nowait files.

Whereas the SETMODE procedure is a waited operation and suspends the caller while waiting for a request to complete, the SETMODENOWAIT procedure returns to the caller after initiating a request. A call to SETMODENOWAIT completes in a call to AWAITIO[X]. The *count-transferred* parameter to AWAITIO[X] has no meaning for SETMODENOWAIT completions. The *buffer-addr* parameter is set to the address of *last-params* parameter of SETMODENOWAIT.

Syntax for C Programmers

```
#include <cextdecs(SETMODENOWAIT)>

_cc_status SETMODENOWAIT ( short filenum
                          ,short function
                          ,[ short param1 ]
                          ,[ short param2 ]
                          ,[ short _near *last-params ]
                          ,[ __int32_t tag ] );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.
- The function value returned by SETMODENOWAIT, which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL SETMODENOWAIT ( filenum                    ! i
                    ,function                   ! i
                    ,[ param1 ]                 ! i
                    ,[ param2 ]                 ! i
                    ,[ last-params ]            ! o
                    ,[ tag ] );                 ! i
```

Parameters

filenum input

INT:value

is a number of an open file, identifying the file to receive the SETMODENOWAIT *function*.

function input

INT:value

is one of the device-dependent functions listed in Table 14-4 on page 14-63 (see SETMODE Procedure on page 14-60).

param1 input

INT:value

is one of the *param1* values listed in Table 14-4 on page 14-63 (see SETMODE Procedure on page 14-60). If omitted, for a disk file the present value is retained. For SETMODEs on other devices, this value depends on the device and the value supplied in the *function* parameter.

param2 input

INT:value

is one of the *param2* values listed in Table 14-4 on page 14-63 (see SETMODE Procedure on page 14-60). If omitted, for a disk file the present value is retained. For SETMODEs on other devices this value depends on the device and the value supplied in the *function* parameter.

last-params output

INT:ref:2

returns the previous settings of *param1* and *param2* associated with the current *function*. The format is:

last-params[0] = old *param1*
last-params[1] = old *param2* (if applicable)

tag input

INT(32):value

is for nowait I/O only. *tag* is a value you define that uniquely identifies the operation associated with this SETMODENOWAIT.

Note. The system stores the *tag* value until the I/O operation completes. The system then returns the *tag* information to the program in the *tag* parameter of the call to AWAITIO[X], thus indicating that the operation completed.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILE_GETINFO_ or FILEINFO).

= (CCE) indicates that the SETMODENOWAIT is successful.

> (CCG) indicates that the SETMODENOWAIT function is not allowed for this device type.

Considerations

- File opened with wait depth > 0

  AWAITIO[X] must be called to complete the call when *filenum* is opened with a wait depth greater than 0. For files with wait depth equal to 0, a call to SETMODENOWAIT is a waited operation and performs in the same way as a call to SETMODE.

- *last-params* and AWAITIO[X]

  AWAITIO returns @*last-params* in the *buffer* parameter (AWAITIOX returns the extended address of *last-params*). The count is undefined.

- For disk files, a call to SETMODENOWAIT is a waited operation and it is performed in the same way as a call to SETMODE.

Example

```
LITERAL SET^SPACE = 6,
        NO^SPACE = 0,
        SPACE = 1;
          .
          .
CALL SETMODENOWAIT ( FILE^NUM , SET^SPACE , SPACE );
     ! turns off single spacing for a line printer.
```

Related Programming Manuals

For programming information about the SETMODENOWAIT file-system procedure, see the *Guardian Programmer's Guide* and the data communication manuals.

SETMYTERM Procedure (Superseded by PROCESS_SETSTRINGINFO_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The SETMYTERM procedure permits a process to change the terminal that is used as its home terminal (the default home terminal is the home terminal of a process's creator).

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
CALL SETMYTERM ( terminal-name );                    ! i
```

Parameters

terminal-name input

INT:ref:12

contains the internal-format file name of the terminal or the process that is to function as the caller's home terminal.

Condition Code Settings

< (CCL) indicates that the terminal cannot be reassigned, *terminal-name* is invalid, *terminal-name* is not a terminal or a named process, or *terminal-name* has a second level qualifier (as in the example, $TERM.#Q1.Q2).

= (CCE) indicates that the SETMYTERM is successful.

> (CCG) does not return from SETMYTERM.

Considerations

If the caller to SETMYTERM creates any processes after the call to SETMYTERM, the new home terminal is the home terminal for those processes. SETMYTERM has no effect on any existing process created by the caller.

SETPARAM Procedure

Summary

The SETPARAM procedure is used to set and fetch various values such as the station characteristics of network addresses. The operation can be performed in a nowait manner by use of the *nowait-tag* parameter and in that case is completed by a call to AWAITIO[X].

Syntax for C Programmers

```
#include <cextdecs(SETPARAM)>

_cc_status SETPARAM ( short filenum
                     ,short function
                     ,[ short _near *param-array ]
                     ,[ short param-count ]
                     ,[ short _near *last-param-array ]
                     ,[ short _near *last-param-count ]
                     ,[ short last-param-max ]
                     ,[ __int32_t nowait-tag ] );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.
- The function value returned by SETPARAM, which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL SETPARAM ( filenum                    ! i
               ,function                   ! i
               ,[ param-array ]            ! i
               ,[ param-count ]            ! i
               ,[ last-param-array ]       ! o
               ,[ last-param-count ]       ! o
               ,[ last-param-max ]         ! i
               ,[ nowait-tag ] );          ! i
```

Parameters

filenum input

INT:value

is the number of an open file for which special information is sent.

function input

INT:value

is one of these SETPARAM function codes:

1	Set or fetch a remote data terminal equipment address (use with X.25 Access Method (X25AM) only).
2	Set or fetch the clear cause or diagnostic bytes (use with X25AM only).
3	Set or fetch parameters for BREAK handling.
4	Set or fetch the reset cause or diagnostic bytes (use with X25AM only).
5	Fetch the restart cause or diagnostic bytes (use with X25AM only).
6	Set or fetch the 6520 and 6530 block mode terminal error counters (use with interactive terminal interface (ITI) protocol in X25AM only).
7	Set or override the closed user's group (CUG) number to be used in next call request packet.
8	Set or fetch the protocol ID field in the outgoing call request packet (use with process-to-process protocol in X25AM only).
9	Fetch the reason why circuit disconnected and learn the current link status (use with X25AM only).
20	Reset and retrieve the called data terminal equipment (DTE) address buffer.
21	Provide a count of the number of 64-byte segments that can be sent and received by a subdevice.
22	Access the Level 4 ITI protocol block mode timer.
153	Fetch the 4-byte SNA sense code and the 4-byte exception response identification number (use only when SNAX exception response mode is enabled).

For a detailed description of function 3, see the *Device-Specific Access Methods - AM3270/TR3271* and the *Device-Specific Access Method - AM6520* manuals, and the *Asynchronous Terminal and Printer Processes Programming Manual*. For a detailed description of functions 1, 2, 4, 5, 6, 8, and 9, see the *X.25 Access Method (X25AM) Manual*. For a detailed description of function 153, see the *SNAX/XF Application Programming Manual.*

param-array input

INT:ref:*

is a list or string as required by *function*.

param-count input

INT:value

is the number of bytes contained in *param-array*.

last-param-array output

INT:ref:*

returns previous parameter settings associated with *function*.

last-param-count output

INT:ref:1

returns the length in bytes of the data placed into *last-param-array*.

last-param-max input

INT:value

is the maximum number of bytes that can be placed in *last-param-array*. The default is 256.

nowait-tag input

INT(32):value

if present and not -1D, specifies that the operation is to be performed in a nowait manner and specifies the value to be returned in the *tag* parameter of AWAITIO[X] at completion. If *nowait-tag* is omitted, or is -1D, or the file has a nowait depth of 0, the operation is waited and is complete when the procedure returns. See Considerations.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILE_GETINFO_ or FILEINFO).

= (CCE) indicates that the SETPARAM is successful.

> (CCG) indicates that the SETPARAM function is not allowed for this device type.

Considerations

- *param-array* and *last-param-array* (applies only for *function* 3).

  These are integer arrays containing:

  word [0] equivalent to parameter 1 of SETMODE 11

[1]	equivalent to parameter 2 of SETMODE 11
[2]	most significant word of the break tag
[3]	least significant word of the break tag

where:

[0]	0, disable BREAK (default setting) 1, enable BREAK and take ownership value from *last-param-array*[1], return BREAK ownership to previous owner
[1]	Terminal access mode after BREAK is typed. 0, normal mode (any file-type access is permitted) 1, break mode (only break-type file access permitted)
[2:3]	are the two words of the 32-bit tag. This is saved by the I/O process handling the terminal. Whenever the I/O process detects that BREAK has been typed, a break-on-device message is sent to the owner of the BREAK for that terminal. (The owner of BREAK is specified by the parameter in *buf*[0].) For details, see the descriptions of the break-on-device messages in the *Guardian Procedure Errors and Messages Manual*.

- Processes

  The SETPARAM procedure can be called on an opened process if the receiving process has indicated (by the system messages flag of the FILE_OPEN_ or OPEN call and by setmode 80) that it is willing to accept messages for such calls. The system message that is delivered and used for reply is the process SETPARAM message (-37). For the format of this message, see the *Guardian Procedure Errors and Messages Manual*.

- SETPARAM in a nowait manner

  When the SETPARAM operation is performed in a nowait manner (a value other than -1D is supplied for the *nowait-tag* parameter), it must be completed by a call to AWAITIO[X]. The *buffer-addr* parameter of AWAITIO[X] is set to the address of the *last-param-array* parameter of SETPARAM; the *count-transferred* parameter of AWAITIO[X] returns the number of bytes returned in *last-param-array*.

Example

```
CALL SETPARAM ( FNUM , 8 , , , OLD^PARAMS , OLD^SIZE );
```

The above example fetches the protocol ID field in the outgoing call request packet. Four bytes of data return.

Related Programming Manuals

For programming information about the SETPARAM procedure, see the data communication manuals.

SETSTOP Procedure

Summary

The SETSTOP procedure permits a process to protect itself from being deleted by any process other than itself or its creator.

Syntax for C Programmers

```
#include <cextdecs(SETSTOP)>

short SETSTOP ( short stop-mode );
```

Syntax for TAL Programmers

```
last-stop-mode :=  SETSTOP ( stop-mode );        ! i
```

Parameters

last-stop-mode returned value

INT

returns a value that is either the preceding value of *stop-mode* or -1 if an invalid mode was specified.

stop-mode input

INT:value

is a value specifying a new stop mode. The modes are:

0 Stoppable by any process. This mode is never set by any system software; it must be set by the user with this procedure specifically for an application. This mode cannot be set by an OSS process.

1 stoppable only by (normal system default value)

- The super ID

- A process whose process access ID = this process's creator access ID (CAID) or the CAID group manager
- A process whose process access ID = this process's process access ID (PAID) or the PAID group manager (this includes the caller to STEPMOM)

2 unstoppable by any other process. This mode is available only when the caller of SETSTOP is privileged.

For additional information about the super ID and process access ID, see the description of the PROCESS_GETINFO_ procedure for information on the two access IDs, as well as to the *Guardian User's Guide*. For additional information on stopping a process, see the description of the PROCESS_STOP_ Procedure or STOP Procedure (Superseded by PROCESS_STOP_ Procedure) procedure.

Considerations

- The default stop mode is 1 when a process is created.
- If a process's stop mode is 1 when a PROCESS_STOP_ or STOP procedure call is issued against it by a process without the authority to stop it, the process does not stop; the process is deleted, however, if and when the stop mode is changed to 0.
- If a process's stop mode is 2 when a PROCESS_STOP_ or STOP procedure call is issued against it by another process, the stop is queued until the process is in a stoppable mode.
- If a process's stop mode is 2 when an unhandled trap or signal occurs, it causes a processor halt. Such a halt occurs, for example, if an unmirrored disk volume that the process is using as a swap volume goes down.

OSS Considerations

An OSS process can be stopped only according to the rules specified for the OSS `kill()` function. Stop mode 0 is therefore not allowed for OSS processes. For details, see the `kill(2)` function reference pages either online or in the *Open System Services System Calls Reference Manual*.

Example

```
LAST^MODE := SETSTOP ( NEW^MODE );
```

Related Programming Manual

For programming information about the SETSTOP procedure, see the *Guardian Programmer's Guide*.

SETSYNCINFO Procedure (Superseded by FILE_SETSYNCINFO_ Procedure)

Summary

The SETSYNCINFO procedure is used by the backup process of a process pair after a failure of the primary process.

The SETSYNCINFO procedure passes a process pair's latest synchronization block (received in a checkpoint message from the primary) to the file system. Following a call to the SETSYNCINFO procedure, the backup process can retry the same series of write operations started by the primary before its failure. The use of the sync block ensures that operations which might have been completed by the primary before its failure are not duplicated by the backup.

Note. Typically, SETSYNCINFO is not called directly by application programs. Instead, it is called indirectly by CHECKMONITOR.

Syntax for C Programmers

```
#include <cextdecs(SETSYNCINFO)>

_cc_status SETSYNCINFO ( short filenum
                        ,short _near *sync-block );
```

- The function value returned by SETSYNCINFO, which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL SETSYNCINFO ( filenum                          ! i
                  ,sync-block );                    ! i
```

Parameters

filenum input

INT:value

is the number of an open file that identifies the file whose synchronization block is being passed.

sync-block input

INT:ref:*

is the latest synchronization block received from the primary process.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILE_GETINFO_ or FILEINFO).

= (CCE) indicates that SETSYNCINFO is successful.

> (CCG) indicates that the file is not a disk file.

Considerations

- The SETSYNCINFO procedure cannot be used with Enscribe format 2 files or OSS files larger than approximately 2 gigabytes. If an attempt is made to use the SETSYNCINFO procedure with these files, error 581 is returned. For information on how to perform the equivalent task with Enscribe format 2 files or OSS files larger than approximately 2 gigabytes, see the FILE_SETSYNCINFO_ Procedure.
- File number has not been opened

  If the SETSYNCINFO file number does not match the file number of the open file, then the call to SETSYNCINFO is rejected with file-system error 16.
- Application parameter or buffer address out of bounds

  If an out-of-bounds application buffer address parameter is specified in the SETSYNCINFO call (that is, a pointer to the buffer has an address that is outside of the data area of the process), then the call is rejected with file-system error 22.
- Checksum error on file sync block

  If an attempt is made to modify the file-system sync buffer area, the SETSYNCINFO call is rejected with file-system error 41.

Example

```
CALL SETSYNCINFO ( F1 , SYNC );
```

SETSYSTEMCLOCK Procedure

Summary

The SETSYSTEMCLOCK procedure allows you to change the system clock if you are a member of the super group.

Syntax for C Programmers

```
#include <cextdecs(SETSYSTEMCLOCK)>

_cc_status SETSYSTEMCLOCK ( long long julian-gmt
                           ,short mode
                           ,short tuid );
```

- The function value returned by SETSYSTEMCLOCK, which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL SETSYSTEMCLOCK ( julian-gmt              ! i
                     ,mode                    ! i
                     ,[ tuid ] );             ! i
```

Parameters

julian-gmt input

FIXED:value

is the Julian timestamp.

mode input

INT:value

specifies the mode and source as follows:

Mode		Source	Action
0	Conditionally adjust to absolute Greenwich mean time (GMT)	operator input	If the clock error is less than or equal to two minutes, the system adjusts the clock. Otherwise the system sets the clock.
1	Conditionally adjust to absolute GMT	hardware clock	If the clock error is less than or equal to two minutes, the system adjusts the clock. Otherwise the system sets the clock.
2	Conditionally adjust to relative GMT	operator input	If the clock error is less than or equal to two minutes, the system adjusts the clock. Otherwise the system sets the clock.
3	Conditionally adjust to relative GMT	hardware clock	If the clock error is less than or equal to two minutes, the system adjusts the clock. Otherwise the system sets the clock.
5	Set relative GMT	operator input	Regardless of the clock error, the system sets the clock.
6	Adjust to relative GMT regardless of clock error	operator input	Regardless of the clock error, the system adjusts the clock.
7	Set absolute GMT	operator input	Regardless of the clock error, the system sets the clock.
8	Stop adjustment	not applicable	Any clock adjustments are stopped.

Absolute mode means that the *julian-gmt* parameter contains the actual time to which you want to set the system clock.

Relative mode means that the *julian-gmt* parameter contains the microsecond correction by which you want to change the system clock; it is not an actual

timestamp. This mode is used for precise time synchronization with a hardware clock or for a moderately precise method of operator time adjustment.

tuid input

INT:value

is a time update ID obtained from the JULIANTIMESTAMP procedure. It should be used with *mode* 2 and 3 to avoid conflicting changes.

Condition Code Settings

< (CCL) indicates one of these:

- insufficient capability
- time specified by user is out of range (01 JAN 1975 0:00:00.000000 to 31 DEC 10000 23:59:59.999999)
- *julian-gmt* is not supplied; invalid *mode*; or *tuid* does not match.
- requested adjustment exceeds one hour and *mode* specified is 6

= (CCE) indicates that the SETSYSTEMCLOCK was successful.

> (CCG) is not returned by SETSYSTEMCLOCK.

Automatically Adjusting the Clock Using Modes 0,1,2,3,6

- If the value of the *mode* parameter is 0 through 3 and the clock error is two minutes or less (as determined from the user input), the system adjusts the clock rather than sets it. If the value of the *mode* parameter is 6, then regardless of the clock error, the system adjusts the clock rather than sets it. The system adjusts the clock by very small amounts. For example, if the clock is slow, making a change of two minutes takes about 33 hours; if the clock is fast, making a change of two minutes takes about 14 days.
- When making an adjustment, SETSYSTEMCLOCK tries to determine the clock rate error by relating the clock error to the time elapsed since the clock was last set.
- If you call SETSYSTEMCLOCK, if the *mode* parameter is not 6, and if the clock error is greater than two minutes, then the system stops any ongoing adjustment and sets the clock.
- If you call SETSYSTEMCLOCK with a *mode* parameter value of 6, and the requested adjustment exceeds one hour, CCL is returned.

Setting the Clock Using Modes 5 and 7

- If the value of the mode parameter is 5 or 7, regardless of the clock error, the system sets the clock.

Stopping Clock Adjustment

- If you call SETSYSTEMCLOCK with the mode parameter set to 8, the system stops any ongoing adjustment.
- If you call SETSYSTEMCLOCK twice in less than ten seconds, the system stops any ongoing adjustment and sets the clock.
- HP reserves the right to change, with proper customer notification, the characteristics of system clock setting and adjustment. The only guaranteed feature is that calling SETSYSTEMCLOCK twice in ten seconds causes the clock to be set. The only exception is when the *mode* parameter has the value 6. In this case, the clock will not be set under any circumstances.

Types of Timestamps

- A 48-bit timestamp is a quantity equal to the number of 10-millisecond units since 00:00, 31 December 1974. The 48-bit timestamp always represents local civil time.
- Procedures that work with the 48-bit timestamp are CONTIME, TIME, and TIMESTAMP.
- A 64-bit Julian timestamp is based on the Julian Date. It is a quantity equal to the number of microseconds since January 1, 4713 B.C., 12:00 (noon) Greenwich mean time (Julian proleptic calendar). This timestamp can represent either Greenwich mean time, local standard time, or local civil time. There is no way to examine a Julian timestamp and determine which of the three times it represents.
- Procedures that work with the 64-bit Julian timestamp are COMPUTETIMESTAMP, CONVERTTIMESTAMP, INTERPRETTIMESTAMP, JULIANTIMESTAMP, and SETSYSTEMCLOCK.
- All time and calendar units in this discussion are defined in *The Astronomical Almanac* published annually by the U.S.þNaval Observatory and the Royal Greenwich Observatory.

Timing and Processes

- Process creation time is initialized by calling TIMESTAMP, which returns the local civil time in centiseconds (0.01 second = 10 milliseconds) since midnight (00:00) on 31 December 1974 in an array of three words. Only the two low-order words are saved in the process control block (PCB); this is sufficient to make the unnamed process ID unique.
- Process timing uses 64-bit elapsed time counters with microsecond resolution; these are not Julian timestamps either.
- There is no way to generalize about internal timing using 64-bit Julian timestamps or 48-bit timestamps. Each section of the operating system manages time using the method most appropriate for its application.

System Clock System Message

- The SETTIME system message (-10) is delivered to the calling process if the MONITORNEW procedure has enabled it.

Example

```
CALL SETSYSTEMCLOCK ( JULIAN^GMT , MODE , TUID );
```

Related Programming Manual

For programming information about the SETSYSTEMCLOCK procedure, see the *Guardian Programmer's Guide*.

SHIFTSTRING Procedure (Superseded by STRING_UPSHIFT_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The SHIFTSTRING procedure upshifts or downshifts all alphabetic characters in a string. Nonalphabetic characters remain unchanged.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
CALL SHIFTSTRING ( string                ! i, o
                 ,count                  ! i
                 ,casebit );             ! i
```

Parameters

string input, output

STRING:ref:*

is the character string to be shifted.

count input

INT:value

is the length of the string in bytes.

casebit input

INT:value

specifies a value indicating whether to upshift or downshift the string:

<15> 0 the procedure upshifts the string indicated, making all alphabetic characters uppercase.

<15> 1 the procedure downshifts the string indicated, making all alphabetic characters lowercase.

Example

```
CALL SHIFTSTRING (COMMAND , COMMAND^LEN , CASE^BIT);
                                          ! upshift
```

Related Programming Manual

For programming information about the SHIFTSTRING utility procedure, see the *Guardian Programmer's Guide*.

SIGACTION_ Procedure

Note. This procedure can be called only from native processes.

SIGACTION_ is the pTAL procedure name for the C `sigaction()` function. The C `sigaction()` function complies with the POSIX.1 standard.

For the pTAL prototype definitions, see the $SYSTEM.SYSTEM.HSIGNAL header file. For a discussion of each parameter and procedure considerations, see the `sigaction(2)` function reference page either online or in the *Open System Services System Calls Reference Manual*.

SIGACTION_INIT_ Procedure

Summary

Note. This procedure can be called only from native processes.

The SIGACTION_INIT_ procedure establishes the caller's initial state of signal handling if default handling is not desired.

Syntax for C Programmers

```
#include <tdmsig.h>

_int32_t SIGACTION_INIT_ ( void (* handler) ( int signum
                                               ,siginfo_t *
                                               ,void * ) );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
?SOURCE $SYSTEM.SYSTEM.HTDMSIG

error := SIGACTION_INIT_ ( handler );                    ! i
```

Parameters

error returned value

INT(32)

indicates the outcome of the call:

0D indicates a successful outcome.

-1D indicates an error. The reason for the error is given in the `errno` variable. Use the ERRNO_GET_ procedure to obtain the value of `errno` in a pTAL program.

handler input

PROCADDR:ref:1

specifies the action to be invoked when a signal occurs. Its value can be:

- The address of a native procedure to perform the signal handling
- SIG_DFL to install the default action
- SIG_ABORT to abnormally terminate the process
- SIG_DEBUG to enter debugging state

For details, see Handler Considerations on page 14-124.

General Considerations

- This procedure is the functional equivalent the TNS ARMTRAP procedure for native processes.
- POSIX.1 compliance

  This procedure is an extension to the POSIX.1 standard. The same effect can be achieved while maintaining compliance with the POSIX.1 standard by calling the SIGPROCMASK_ procedure and a loop of SIGACTION_ procedure calls.
- Calling considerations

  The SIGACTION_INIT_ procedure is designed to be called once, typically from the main procedure of a program. Although it is not an error to call this procedure twice, native Guardian C programmers should be aware that the CRE makes this call before invoking the program's main function.

SIGACTION_INIT_ sets the signal mask to unblock all signals. Pending signals are discarded.

The specified handler (or action) is installed for all signals whose default action is not SIG_IGN and for which it is valid to specify the associated action. (You cannot specify an action for the OSS SIGKILL, SIGABEND, or SIGSTOP signal.) The action for any signal that is ignored by default is set to SIG_IGN. If the action specified is not applicable to a specific signal, then the existing action for that signal remains unchanged.

If SIGACTION_INIT_ returns an error, the signal-handling state is not changed.

- Deferrable and nondeferrable signals

  Deferrable signals that occur while the process is executing privileged code are deferred until the process exits privileged execution mode. If these signals are not blocked, they are then delivered to the handler, which is activated at the tip of the main stack.

  Nondeferrable signals are immediately delivered to the specified handler. The handler executes on the main stack or privileged stack of the calling process depending on whether the signal occurs in nonprivileged code or privileged code and on whether the signal handler for that signal is installed by a nonprivileged caller or privileged caller of SIGACTION_INIT_ as follows:

If a nondeferrable signal occurs in...	And the signal handler for that signal was installed by...	Then the specified handler is activated at...
nonprivileged code	a nonprivileged or privileged caller of SIGACTION_INIT_	the tip of the main stack (when the signal was generated)
privileged code	a nonprivileged caller of SIGACTION_INIT_	the tip of the main stack (when the process entered privileged mode)
privileged code	a privileged caller of SIGACTION_INIT_	the tip of the privileged stack

- Signal mask and nondeferrable signals

  Before a signal handler is entered, a new signal mask is installed. This mask is formed from the union of the current signal mask and the signal being delivered. If a nondeferrable signal occurs and is blocked, the process abnormally terminates.

Note. Abnormally terminating a process when a nondeferrable signal is blocked is an extension to the POSIX.1 standard. According to the POSIX.1 standard, a blocked nondeferrable signal has an undefined outcome.

Handler Considerations

- *handler* must be one of these:
  - The address of an untyped native procedure that accepts these three parameters. These parameters are passed to the handler by the system when the handler is invoked to catch a signal:
    - SIGNUM

      An INT(32) numeric value indicating the signal that caused the handler to be invoked.
    - SIGINFO

      A pointer whose value is currently NULL.
    - UCONTEXT

      A pointer to a structure of type UCONTEXT_T. It contains information regarding the process context when the signal occurred. You can pass this pointer to the HIST_INIT_ procedure to get diagnostic information.
  - SIG_DFL

    Causes default signal handling to be installed.
  - SIG_ABORT

    Causes the process to be abnormally terminated when a signal occurs.

    Note. This action is similar to calling ARMTRAP(-1,-1) for a TNS Guardian process.
  - SIG_DEBUG

    Causes the process to enter debug mode when a signal occurs.

Note. The SIG_ABORT and SIG_DEBUG options are HP extensions to the POSIX.1 standard.

- If the signal was generated as a nondeferrable signal, the signal handler should not execute a simple return; otherwise, process termination results. You must exit the signal handler using either the SIGLONGJMP_ or LONGJMP_ procedure. SIGLONGJMP_ is preferred, because it allows you to restore the signal mask that was saved by the corresponding SIGSETJMP_ procedure, so your process can receive multiple occurrences of the same nondeferrable signal. LONGJMP_ does not restore the signal mask; therefore the signal that was handled remains blocked.

  For a deferrable signal, the signal handler can simply return, causing process execution to resume where it was preempted by the signal.

Examples

TAL example:

```
error := SIGACTION_INIT_ ( @SIG_HANDLER );
IF error = <>0 THEN
   errnoval := ERRNO_GET_;
```

C example:

```
if (SIGACTION_INIT_ ( myhandler ) != 0)
    /* handle error */
```

Related Programming Manual

For programming information about the SIGACTION_INIT_ procedure, see the *Guardian Programmer's Guide*.

SIGACTION_RESTORE_ Procedure

Summary

Note. This procedure can be called only from native processes.

The SIGACTION_RESTORE_ procedure restores the signal-handling state saved by a previous call to the SIGACTION_SUPPLANT_ procedure. SIGACTION_SUPPLANT_ allows a subsystem (such as a shared run-time library) to take over signal handling temporarily. SIGACTION_RESTORE_ restores the signal-handling state of the process before the subsystem exits.

Syntax for C Programmers

```
#include <tdmsig.h>

int SIGACTION_RESTORE_ ( sig_save_template *signal-buffer );
```

Syntax for TAL Programmers

```
?SOURCE $SYSTEM.SYSTEM.HTDMSIG

error := SIGACTION_RESTORE_ ( signal-buffer );          ! i
```

Parameters

error returned value

INT(32)

indicates the outcome of the call:

0D indicates a successful outcome.

-1D indicates an error. The reason for the error is given in the `errno` variable:

`FE_EFAULT` The address in *signal-buffer* is invalid.

`FE_EINVAL` the content of the *signal-buffer* contains invalid data or SIGACTION_SUPPLANT_ was not called.

Use the ERRNO_GET_ procedure to obtain the value of `errno` in a Guardian process.

signal-buffer input

INT .EXT:ref:(SIG_SAVE_TEMPLATE)

specifies the address of a buffer in which the previous signal-handling state is saved.

Considerations

- The SIGACTION_RESTORE_ procedure validates the buffer indicated by *signal-buffer* and returns an error if the buffer is invalid. It is assumed that the buffer has been initialized by the SIGACTION_SUPPLANT_ procedure and that it has not been modified by the caller.
- The signal-handling state previously saved in the buffer indicated by *signal-buffer* is atomically restored.

- Any pending signal that is unblocked by restoring the saved signal mask will be delivered to the restored signal handler after exiting this procedure or returning to user code.
- A privileged caller is allowed to restore nonprivileged signal-handling specifications. If the caller is nonprivileged, however, the restored signal handling state is marked as nonprivileged.

Example

```
error := SIGACTION_RESTORE_ ( buffer );
IF error <> 0 THEN
   errnoval := ERRNO_GET_;
```

Related Programming Manual

For programming information about the SIGACTION_RESTORE_ procedure, see the *Guardian Programmer's Guide*.

SIGACTION_SUPPLANT_ Procedure

Summary

Note. This procedure can be called only from native processes.

The SIGACTION_SUPPLANT_ procedure allows a subsystem (such as a shared run-time library) to take over signal handling temporarily. Before exiting, the same subsystem calls the SIGACTION_RESTORE_ restore to restore the signal-handling state established by the process before entering the subsystem.

Syntax for C Programmers

```
#include <tdmsig.h>

int SIGACTION_SUPPLANT_ ( void (* handler) ( int signum
                                            ,siginfo_t *
                                            ,void * )
                        ,sig_save_template *signal-buffer
                        ,short length );
```

Syntax for TAL Programmers

```
?SOURCE $SYSTEM.SYSTEM.HTDMSIG

error := SIGACTION_SUPPLANT_ ( handler                  ! i
                             ,signal-buffer             ! o
                             ,length );                 ! i
```

Parameters

error returned value

INT(32)

indicates the outcome of the call:

0D indicates a successful outcome

-1D indicates an error. The reason for the error is given in the `errno` variable:

`FE_EFAULT` The address in *signal-buffer* is out of bounds.

`FE_EINVAL` `SIG_IGN` or `SIG_ERR` is passed to the handler.

`FE_ERANGE` *length* is less than the minimum required.

Use the ERRNO_GET_ procedure to obtain the value of `errno` in a Guardian process.

handler input

PROCADDR:value

specifies the action to be invoked when a signal occurs. Its value can be:

- The address of a native procedure to perform the signal handling
- SIG_DFL to accept the default action
- SIG_ABORT to abnormally terminate the process
- SIG_DEBUG to enter debug mode

For details, see Handler Considerations on page 14-131.

signal-buffer output

INT .EXT:ref:*

returns the address of a buffer in which the previous signal-handling state is saved. The buffer is allocated using the SIGSAVE_DEF_ DEFINE.

length input

INT:value

specifies the size in bytes of the buffer indicated by *signal-buffer*.

General Considerations

- POSIX.1 compliance

  This procedure is an extension to the POSIX.1 standard. A similar same effect can be achieved while maintaining compliance with the POSIX.1 standard by calling the SIGPROCMASK_ procedure and the SIGACTION_ procedure for each signal. However, SIGACTION_SUPPLANT_ also establishes a state in which a deferrable signal can be blocked but the same signal will invoke the handler if generated as nondeferrable.

- Calling considerations

  You must allocate the buffer for SIGACTION_SUPPLANT_ using the SIGSAVE_DEF DEFINE as follows:

  ```
  SIGSAVE_DEF ( signal-buffer );
  ```

  where *signal-buffer* is a valid variable name. This buffer must be accessible to the callers of both SIGACTION_SUPPLANT_ and the associated SIGACTION_RESTORE_ procedure.

  The specified handler is installed as the action for only those nondeferrable signals that are system generated in response to run-time events. These signals are:

  SIGILL
  SIGFPE
  SIGSEGV
  SIGMEMERR
  SIGNOMEM
  SIGMEMMGR
  SIGSTK
  SIGLIMIT

  The signal handling for other signals remains unchanged.

  All parameters are validated. The SIGACTION_SUPPLANT_ procedure returns an error if any parameter has an invalid value.

If SIGACTION_SUPPLANT_ returns an error, the signal-handling state is not changed.

- All signals, except those for which you cannot install a handler, are blocked.

  Deferrable signals that occur while the process is executing privileged code are deferred until the process exits privileged execution mode. If these signals are not blocked, they are then delivered to the handler, which is activated at the tip of the main stack.

  Nondeferrable signals are immediately delivered to the specified handler. The handler executes on the main stack or privileged stack of the calling process depending on whether the signal occurs in nonprivileged code or privileged code and on whether the signal handler for that signal is installed by a nonprivileged caller or privileged caller of SIGACTION_SUPPLANT_ as follows:

If a nondeferrable signal occurs in...	**And the signal handler for that signal was installed by...**	**Then the specified handler is activated at...**
nonprivileged code	a nonprivileged or privileged caller of SIGACTION_SUPPLANT_	the tip of the main stack (when the signal was generated)
privileged code	a nonprivileged caller of SIGACTION_SUPPLANT_	the tip of the main stack (when the process entered privileged mode)
privileged code	a privileged caller of SIGACTION_SUPPLANT_	the tip of the privileged stack

- Nested signals

  Signals can be nested. If a different signal occurs during execution of a signal handler—or any procedure called directly or indirectly from the signal handler—the handler for that signal is invoked at the current tip of the stack.

Note. This action differs from the corresponding action on a TNS Guardian process; a trap that occurs during execution of a trap handler is fatal to the process.

- Signal mask

  SIGACTION_SUPPLANT_ sets the signal mask to block all signals from delivery. All signals that can be deferred are kept pending. Any nondeferrable signal is delivered to the handler.

Note. This response to a nondeferrable signal is an extension to the POSIX.1 standard; according to the POSIX.1 standard, the response to a nondeferrable signal is undefined. This response also differs from the OSS implementation, which abnormally terminates the process if a nondeferrable signal is blocked.

Handler Considerations

- *handler* must be one of these:
  - The address of an untyped native procedure that accepts these three parameters. These parameters are passed to the handler by the system when the handler is invoked to catch a signal:
    - SIGNUM

      An INT(32) numeric value indicating the signal that caused the handler to be invoked.
    - SIGINFO

      A pointer whose value is currently NULL.
    - UCONTEXT

      A pointer to a structure of type UCONTEXT_T. It contains information regarding the process context when the signal occurred. You can pass this pointer to the HIST_INIT_ procedure to get diagnostic information.
  - SIG_DFL

    Causes default signal handling to be installed for all signals.
  - SIG_ABORT

    Causes the process to be abnormally terminated when a signal occurs.

    Note. This action is similar to calling ARMTRAP(-1,-1) for a TNS process.
  - SIG_DEBUG

    Causes the process to enter debug mode when a signal occurs.

  Note. The SIG_ABORT and SIG_DEBUG options are HP extensions to the POSIX.1 standard.
- If the signal was generated as a nondeferrable signal, the signal handler should not execute a simple return; otherwise, process termination results. You must exit the signal handler using either the SIGLONGJMP_ or LONGJMP_ procedure; the effect is the same for either procedure, because the signal mask is set to block all signals regardless of whether the original mask is restored.

  For a deferrable signal, the signal handler can simply return, causing process execution to resume where it was preempted by the signal.

Example

```
SIGSAVE_DEF ( buffer );
error := SIGACTION_SUPPLANT_ ( handler, buffer, length );
```

```
IF error <> 0 THEN
   errnoval := ERRNO_GET_;
```

Related Programming Manual

For programming information about the SIGACTION_SUPPLANT_ procedure, see the *Guardian Programmer's Guide*.

SIGADDSET_ Procedure

SIGDELSET_ Procedure

SIGEMPTYSET_ Procedure

SIGFILLSET_ Procedure

SIGISMEMBER_ Procedure

Note. These procedures can be called only from native processes.

These procedure names are the pTAL names for the corresponding C functions:

Procedure Name	Corresponding C Function
SIGADDSET_	`sigaddset()`
SIGDELSET_	`sigdelset()`
SIGEMPTYSET_	`sigemptyset()`
SIGFILLSET_	`sigfillset()`
SIGISMEMBER_	`sigismember()`

These functions comply with the POSIX.1 standard.

For the pTAL prototype definitions, see the $SYSTEM.SYSTEM.HSIGNAL header file. For a discussion of each parameter and procedure considerations, see the corresponding `sigaddset(3)`, `sigdelset(3)`, `sigemptyset(3)`, `sigfillset(3)`, and `sigismember(3)` function reference pages either online or in the *Open System Services Library Calls Reference Manual*.

SIGJMP_MASKSET_ Procedure

Summary

Note. This procedure can be called only from native processes.

The SIGJMP_MASKSET_ procedure saves a signal mask in a jump buffer that has already been initialized by the SIGSETJMP_ procedure. Thus, you can avoid the overhead of saving the signal mask when you call SIGSETJMP_ and instead apply the mask at a later time before performing a nonlocal goto with the SIGLONGJMP_ procedure. This technique saves setting the signal mask in applications that have many calls to SIGSETJMP_ and few calls to SIGLONGJMP_.

Syntax for C Programmers

```
#include <tdmsig.h>

long sigjmp_maskset ( jmp_buf *env
                     ,sigset_t *signal-mask );
```

Syntax for TAL Programmers

```
?SOURCE $SYSTEM.SYSTEM.HTDMSIG

error := SIGJMP_MASKSET_ ( env                    ! i,o
                          ,signal-mask );         ! i
```

Parameters

error returned value

INT(32)

indicates the outcome of the call:

0D indicates a successful outcome.

-1D indicates an error. The reason for the error is given in the `errno` variable:

`FE_EINVAL` The jump buffer has not been initialized.

Use the ERRNO_GET_ procedure to obtain the value of `errno` in a Guardian process.

env input, output

INT .EXT:ref:(SIGJMP_BUF_TEMPLATE)

contains the address of a jump buffer containing context saved by the SETJMP_ or SIGSETJMP_ procedure to be restored by a subsequent call to the SIGLONGJMP_ procedure.

signal-mask input

INT .EXT:ref:(SIGSET_T)

If not NULL, points to a valid signal mask that is added to the jump buffer indicated by *env*. A subsequent call to the SIGLONGJMP_ procedure restores the context contained in the jump buffer, including the indicated signal mask.

If NULL, causes the signal mask in the jump buffer indicated by *env* to be cleared. A subsequent call to the SIGLONGJMP_ procedure restores the context contained in the jump buffer, including the clear signal mask that unblocks all signals.

Considerations

- This procedure is an extension to the POSIX.1 standard.
- SIGJMP_MASKSET_ is typically called from a signal handler before a SIGLONGJMP_ procedure is executed. Using SIGJMP_MASKSET_ can avoid the overhead of setting the signal mask in the jump buffer in an application with many calls to SIGSETJMP_ and fewer calls to SIGLONGJMP_.
- The buffer pointer *env* is assumed to be valid and initialized by an earlier SETJMP_ or SIGSETJMP_ call. Otherwise, SIGJMP_MASKSET_ returns -1D and `errno` is set to `FE_EINVAL`.
- This procedure overwrites any signal mask that is already in the jump buffer.
- Any invalid address passed to this procedure will cause the system to deliver a nondeferrable system-generated signal to the process.
- Only the SIGLONGJMP_ procedure, not the LONGJMP_ procedure, can be used with a jump buffer modified by SIGJMP_MASKSET_.

Examples

```
error := SIGJMP_MASKSET_ ( env, mask );
```

or

```
INT .EXT NULL := 0D;
 ...
error := SIGJMP_MASKSET_ ( env, NULL);
```

Related Programming Manual

For programming information about the SIGJMP_MASKSET_ procedure, see the *Guardian Programmer's Guide*.

SIGLONGJMP_ Procedure

Summary

Note. This procedure can be called only from native processes.

The SIGLONGJMP_ procedure performs a nonlocal goto. It restores the state of the calling process using context saved in a jump buffer by the SIGSETJMP_ procedure. Control returns to the location of the corresponding SIGSETJMP_ procedure call. The signal mask is also restored if it was saved; all other signal-handling specifications remain unchanged.

Syntax for C Programmers

```
#include <setjmp.h>

void siglongjmp ( sigjmp_buf env
                 ,int  value );
```

Syntax for TAL Programmers

```
?SOURCE $SYSTEM.ZGUARD.HSETJMP

SIGLONGJMP_ ( env                ! i
             ,value );           ! i
```

Parameters

env input

INT .EXT:ref:(SIGJMP_BUF_TEMPLATE)

indicates the address of a previously allocated and initialized jump buffer containing the process context to be restored by this procedure.

value input

INT(32):value

specifies the value to be returned at the destination of the long jump; that is, at the location of the corresponding SIGSETJMP_ call. If this value is set to 0D, then 1D is returned; otherwise *value* is returned.

Considerations

- SIGLONGJMP_ is the TAL or pTAL procedure name for the C sig`longjmp()` function. The C `siglongjmp()` function complies with the POSIX.1 standard.
- SIGLONGJMP_ does not return. Normally, return is made through the corresponding SIGSETJMP_ procedure.
- Restoring the signal mask with this procedure enables a native process to receive multiple occurrences of the same nondeferrable signal when this procedure is used to exit a signal handler. If the signal mask is not restored and the same nondeferrable signal occurs a second time, then the process terminates.

  For details on deferrable and nondeferrable signals, see SIGACTION_INIT_ Procedure.
- The buffer pointed to by *env* is assumed to be valid and initialized by an earlier call to SIGSETJMP_. If an invalid address is passed or if the caller modifies the jump

buffer, the result is undefined and could cause the system to deliver a nondeferrable signal to the process.

- If SIGLONGJMP_ detects an error, a SIGABRT or SIGILL signal is raised.
- If SIGLONGJMP_ is passed a jump buffer initialized by SETJMP_, then a simple long jump (without restoring the signal mask) is executed.
- The jump buffer must be accessible to both the long jump procedure call and the associated set jump procedure call.
- The procedure that invoked the corresponding call to SIGSETJMP_ must still be active. That is, the activation record of the procedure that called SIGSETJMP_ must still be on the stack.
- A long jump across a transition boundary between the TNS and native mode environments, in either direction, is not permitted. Any attempt to do so will be fatal to the process.
- A nonprivileged caller cannot jump to a privileged area. Any attempt to do so will be fatal to the process. A privileged caller, however, can execute a long jump across the privilege boundary; privileges are automatically turned off before control returns to the SIGSETJMP_ procedure.
- As a result of optimization, the values of nonvolatile local variables in the procedure that calls SIGSETJMP_ might not be the same as they were when SIGLONGJMP_ was called if the variables are modified between the calls to SIGSETJMP_ and SIGLONGJMP_. C and pTAL programs can declare variables with the volatile type qualifier; this is the only safe way of preserving local variables between calls to SIGSETJMP_ and SIGLONGJMP_. Alternatively, you can make the variables global.

Example

```
SIGLONGJMP_ ( env, value );
```

Related Programming Manual

For programming information about the SIGLONGJMP_ procedure, see the *Guardian Programmer's Guide*.

SIGNAL_ Procedure

Note. This procedure can be called only from native processes.

SIGNAL_ is the pTAL procedure name for the C `signal()` function. The C `signal()` function complies with the POSIX.1 standard.

For the pTAL prototype definitions, see the $SYSTEM.SYSTEM.HSIGNAL header file. For a discussion of each parameter and procedure considerations, see the

`signal(3)` function reference page either online or in the *Open System Services Library Calls Reference Manual*.

SIGNALPROCESSTIMEOUT Procedure

Summary

The SIGNALPROCESSTIMEOUT procedure sets a timer based on process execution time, as measured by the processor clock. When the time expires, the calling process receives an indication in the form of a system message on $RECEIVE.

Syntax for C Programmers

```
#include <cextdecs(SIGNALPROCESSTIMEOUT)>

_cc_status SIGNALPROCESSTIMEOUT ( __int32_t timeout-value
                                 ,[ short param1 ]
                                 ,[ __int32_t param2 ]
                                 ,[ short _near *tag ] );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.
- The function value returned by SIGNALPROCESSTIMEOUT, which indicates the condition code, can be interpreted by the `_status_lt()`, `_status_eq()`, or `_status_gt()` function (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL SIGNALPROCESSTIMEOUT ( timeout-value            ! i
                          ,[ param1 ]                ! i
                          ,[ param2 ]                ! i
                          ,[ tag ] );                ! o
```

Parameters

timeout-value input

INT(32):value

specifies the time period, in 0.01-second units, after which a timeout message should be queued on $RECEIVE. This value must be greater than 0D.

param1 input

INT:value

identifies the timeout message read from $RECEIVE.

param2 input

INT(32):value

identifies the timeout message read from $RECEIVE (same purpose as *param1*).

tag output

INT:ref:1

returns an identifier associated with the timer. This *tag* should be used only to call the CANCELPROCESSTIMEOUT procedure.

Condition Code Settings

< (CCL) indicates that SIGNALPROCESSTIMEOUT is unable to allocate a time-list element (TLE).

= (CCE) indicates that SIGNALPROCESSTIMEOUT is successful.

> (CCG) indicates that the given timeout value is invalid or that there is a bounds error on *tag*.

Considerations

- SIGNALPROCESSTIMEOUT and CANCELPROCESSTIMEOUT

  A process can use the SIGNALPROCESSTIMEOUT procedure with the CANCELPROCESSTIMEOUT procedure to verify that some programmatic operation finishes within a certain process execution time. The process calls SIGNALPROCESSTIMEOUT before initiating the operation and then periodically

reads $RECEIVE to watch for timer expiration. The process calls CANCELPROCESSTIMEOUT after completion of the timed operation if the process has not been signaled on $RECEIVE.

- Measuring the time that a process is executing

  The SIGNALPROCESSTIMEOUT procedure measures the time that the process is executing user code and system code. This procedure excludes the time spent by the processor processing interrupts while the process is running.

- Deadlock possibility

  Consider this:

```
CALL SIGNALPROCESSTIMEOUT (10000D,,,TAG);
CALL READ (REC^NUM, BUFFER, 4);
            .
            .  ! open number of $RECEIVE.
```

  The read causes the process to stop and wait for the system message to be generated by timeout (assuming no other messages are expected). Timeout does not occur because process time does not advance while the read is waiting, so a deadlock occurs.

Note. This deadlock does not happen with the SIGNALTIMEOUT procedure, which measures elapsed time (as measured by the processor clock) rather than process execution time.

Message

- Timeout message

  When a time-list element (TLE) set by a call to the SIGNALPROCESSTIMEOUT procedure times out, a system message -26 (process time timeout) is placed on the $RECEIVE queue to be read by the caller. (This message is identical to the message generated by SIGNALTIMEOUT except that the message number is different.)

OSS Considerations

OSS processes can use this procedure and generate a system message. An OSS signal is not generated.

Example

```
CALL SIGNALPROCESSTIMEOUT( VALUE , MSG, , TIMERTAG );
```

Related Programming Manual

For programming information about the SIGNALPROCESSTIMEOUT procedure, see the *Guardian Programmer's Guide*.

SIGNALTIMEOUT Procedure

Summary

The SIGNALTIMEOUT procedure sets a timer to a given number of units of elapsed time, as measured by the processor clock. When the time expires, the calling process receives an indication in the form of a system message on $RECEIVE.

Syntax for C Programmers

```
#include <cextdecs(SIGNALTIMEOUT)>

_cc_status SIGNALTIMEOUT ( __int32_t time-out-value
                          ,[ short param1 ]
                          ,[ __int32_t param2 ]
                          ,[ short _near *tag ] );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.
- The function value returned by SIGNALTIMEOUT, which indicates the condition code, can be interpreted by the `_status_lt()`, `_status_eq()`, or `_status_gt()` function (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL SIGNALTIMEOUT ( timeout-value                ! i
                    ,[ param1 ]                   ! i
                    ,[ param2 ]                   ! i
                    ,[ tag ] );                   ! o
```

Parameters

timeout-value input

INT(32):value

specifies the time period, in 0.01-second units, after which a timeout message should be queued on $RECEIVE. This value must be greater than 0D.

param1 input

INT:value

identifies the timeout message read from $RECEIVE.

param2 input

INT(32):value

identifies the timeout message read from $RECEIVE (same purpose as *param1*).

tag output

INT:ref:1

returns an identifier associated with the timer. This *tag* should be used only to call the CANCELTIMEOUT procedure.

Condition Code Settings

< (CCL) indicates that SIGNALTIMEOUT is unable to allocate a time list element (TLE). This can occur if fewer than one-fourth of the TLEs are free.

= (CCE) indicates that SIGNALTIMEOUT completed successfully.

> (CCG) indicates that the given timeout value is invalid.

Considerations

- SIGNALTIMEOUT and CANCELTIMEOUT

  A process can use the SIGNALTIMEOUT procedure with the CANCELTIMEOUT procedure to verify that some programmatic operation finishes within a certain elapsed time. The process calls SIGNALTIMEOUT before initiating the operation and then periodically reads $RECEIVE to watch for timer expiration. The process calls CANCELTIMEOUT after completion of the timed operation if the process has not been signaled on $RECEIVE.

- Measuring elapsed time that a process executes

  The SIGNALTIMEOUT procedure measures elapsed time (according to the processor clock) that this process executes. This includes the time spent by the processor in process code, in system code, processing interrupts that occur while the process is running, and any time that the process is waiting.

- Using SIGNALTIMEOUT to measure long time intervals

  The SIGNALTIMEOUT procedure measures time according to the internal clock of the processor in which the calling process is executing. Typically, processor time (that is, time as measured by a particular processor) is slightly different from

system time; it also varies slightly from processor to processor, because all the processor clocks typically run at slightly different speeds. System time is determined by taking the average of all the processor times in the system.

When measuring short intervals of time, the difference between processor time and system time is negligible. However, when measuring long intervals of time (such as several hours or more), the difference can be noticeable. For this reason, it is not recommended that you make just one call to the SIGNALTIMEOUT procedure to measure a long interval of time when you need a precise measurement that is synchronized with system time. Instead, you should use a sequence of two or more calls. (The same applies to other procedures, such as DELAY, that also measure time by a processor clock.)

For example, if you want your application to be notified at a specific system time after a long interval, you can use the SIGNALTIMEOUT procedure to set a timer to expire shortly before the desired time. When the timer expires (that is, when a timeout message is delivered to $RECEIVE), your application can compute the remaining time and set another timer for the short interval that remains.

However, because the possibility of clock discrepancy becomes greater as the interval being timed becomes longer, it would be even safer to measure a long time interval by dividing it into a series of relatively short intervals. One method is to compute the interval between the current time and the desired time and set a timer to expire after half that interval. When the timer expires, compute the remaining time and set another timer to expire after half that interval, and so on, approaching the desired time by progressively smaller steps.

Message

- Timeout message

  When a time-list element (TLE) set by a call to the SIGNALTIMEOUT procedure times out, a system message -22 (elapsed time timeout) is sent to the caller's $RECEIVE.

Note. Because a process must read $RECEIVE to be notified when a timer expires, there can be a significant amount of delay before seeing the notification. For example, if a process is waiting for an I/O operation to finish, or if a process has low priority and is waiting to execute, a significant amount of time might pass before the process can read $RECEIVE. The SIGNALTIMEOUT procedure should not be used in situations where such delays cannot be tolerated.

OSS Considerations

OSS processes can use this procedure and generate a system message. An OSS signal is not generated.

Example

```
CALL SIGNALTIMEOUT( 1000D , , , TIMERTAG );   ! 10 seconds.
```

Related Programming Manual

For programming information about the SIGNALTIMEOUT procedure, see the *Guardian Programmer's Guide*.

SIGPENDING_ Procedure

Note. This procedure can be called only from native processes.

SIGPENDING_ is the pTAL procedure name for the C `sigpending()` function. The C `sigpending()` function complies with the POSIX.1 standard.

For the pTAL prototype definitions, see the $SYSTEM.SYSTEM.HSIGNAL header file. For a discussion of each parameter and procedure considerations, see the `sigpending(2)` function reference page either online or in the *Open System Services System Calls Reference Manual*.

SIGPROCMASK_ Procedure

Note. This procedure can be called only from native processes.

SIGPROCMASK_ is the pTAL procedure name for the C `sigprocmask()` function. The C `sigprocmask()` function complies with the POSIX.1 standard.

For the pTAL prototype definitions, see the $SYSTEM.SYSTEM.HSIGNAL header file. For a discussion of each parameter and procedure considerations, see the `sigprocmask(2)` function reference page either online or in the *Open System Services System Calls Reference Manual*.

SIGSETJMP_ Procedure

Summary

Note. This procedure can be called only from native processes.

The SIGSETJMP_ procedure saves process context in a jump buffer. This context is used when a nonlocal goto is performed by a corresponding call to the SIGLONGJMP_ procedure. Optionally, this procedure also saves the current signal mask.

Syntax for C Programmers

```
#include <setjmp.h>

sigjmp_buf env;

__int32_t sigsetjmp ( sigjmp_buf *env
                     ,int mask );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
?SOURCE $SYSTEM.ZGUARD.HSETJMP

retval := SIGSETJMP_ ( env              ! o
                      ,mask );          ! i
```

Parameters

retval returned value

INT(32)

indicates the outcome of the call:

0D indicates that the SIGSETJMP_ procedure was called directly.

< > 0D indicates that SIGSETJMP_ is returning as a result of a call to the SIGLONGJMP_ procedure. The returned value is specified by SIGLONGJMP_.

env output

INT .EXT:ref:(SIGJMP_BUF_TEMPLATE)

indicates the address of a previously allocated jump buffer in which the process context of the caller is returned. The jump buffer is allocated by the SIGJMP_BUF_DEF DEFINE.

mask input

INT(32):value

specifies whether the current signal mask is also saved in the jump buffer indicated by *env*:

0D specifies that the current signal mask is not to be saved.

< > 0D specifies that the current signal mask is to be saved. This mask is reinstated by a corresponding call to the SIGLONGJMP_ procedure.

Considerations

- SIGSETJMP_ is the TAL or pTAL procedure name for the C `sigsetjmp()` function. The C `sigsetjmp()` function conforms to the POSIX.1 standard.
- You can allocate the jump buffer for SIGSETJMP_ using the SIGJMP_BUF_DEF DEFINE as follows:

  ```
  SIGJMP_BUF_DEF ( env );
  ```

  where *env* is a valid variable name.

  Alternatively, you can allocate the buffer by declaring a structure if type SIGJMP_BUF_TEMPLATE.

  In either case, the buffer must be accessible to both the SIGSETJMP_ procedure call and the associated SIGLONGJMP_ procedure call.
- The jump buffer saved by the SIGSETJMP_ procedure is normally used by a call to the SIGLONGJMP_ procedure. The jump buffer can be used by a call to the LONGJMP_ procedure only if the signal mask is not saved.
- The buffer pointer is assumed to be valid. An invalid address passed to SISETJMP_ will cause unpredictable results and could cause the system to deliver a nondeferrable signal to the process.
- Do not change the contents of the jump buffer. The results of a corresponding SIGLONGJMP_ call are undefined if the contents of the jump buffer are changed.

Example

```
sigjmp_buf env;

SIGJMP_BUF_DEF_ ( env );
retval := SIGSETJMP_ ( env, value );
```

Related Programming Manual

For programming information about the SIGSETJMP_ procedure, see the *Guardian Programmer's Guide*.

SIGSUSPEND_ Procedure

Note. This procedure can be called only from native processes.

SIGSUSPEND_ is the pTAL procedure name for the C `sigsuspend()` function. The C `sigsuspend()` function complies with the POSIX.1 standard.

For the pTAL prototype definitions, see the $SYSTEM.SYSTEM.HSIGNAL header file. For a discussion of each parameter and procedure considerations, see the `sigsuspend(2)` function reference page either online or in the *Open System Services System Calls Reference Manual*.

SSIDTOTEXT Procedure

Summary

Converts internal format subsystem ID to external representation.

Syntax for C Programmers

Note. In the TNS/E environment the CEXTDECS file uses the *int* data type for 32-bit values. This is a change from the TNS and TNS/R environments where CEXTDECS uses the *long* data type for 32-bit values.

```
#include <cextdecs(SSIDTOTEXT)>

short SSIDTOTEXT (short ssid
                          ,char *chars
                            ,[__int32_t *status ]);
```

Syntax for TAL Programmers

```
len := SSIDTOTEXT (ssid              !i
          ,chars                     !o
           [,status]);                    !o
```

Parameters

len

INT

contains the number of characters placed into chars. Zero is returned if one of the errors (0,29), (4,0), or (8,0) occurs. Other errors may prevent obtaining the subsystem name, in which case the subsystem ID is still produced but contains the subsystem number rather than the subsystem name.

ssid

INT .EXT:ref:6

contains the subsystem ID to be converted to displayable form.

chars

STRING .EXT:ref:*

contains the resulting displayable representation of ssid. The length of the string is returned in len. The string will not be longer than 23 characters.

status

INT(32) .EXT:ref:1

contains a status code describing any problem encountered. The codes in this list are obtained by examining the two halves of the INT(32) value.

(0,0) - No error.
(0,x) - problem with calling sequence.
x: 29 - required parameter missing
632 - insufficient stack space
(1,x) - error allocating private segment; x is the ALLOCATESEGMENT error code.
(2,x) - problem opening nonresident template file.
x: >0 - file-system error code
-1 - file code not 844
-2 - file not disk file
-3 - file not key-sequenced
-4 - file has wrong record size
-5 - file has wrong primary key definition.

(3,x) - error reading nonresident template file; x is the file-system error.
(4,0) - invalid value in internal subsystem ID.
(7,x) - error accessing private segment; x is the MOVEX error code.
(8,0) - internal error.

Considerations

The external form of a subsystem ID is:

owner.ss.version or 0.0.0

owner is 1 to 8 letters, digits, or hyphens, the first of which must be a letter; letters are not upshifted so the end user must enter *owner* in the proper case.

ss is either the subsystem number or the subsystem name.

A subsystem number is a string of digits which may be preceded by a minus sign. The value of the number must be between -32767 and 32767.

A subsystem name is 1 to 8 letters, digits, or hyphens, the first of which must be a letter. Letters are not upshifted; the end user must enter the subsystem name in the proper case.

version is either a string of digits which represents a TOSVERSION-format version (Ann) or a value from 0 to 65535.

Examples: HP.PATHWAY.C00
HP.52.0
0.0.0

The 0.0.0 form is used to represent the “null” subsystem ID. Its internal representation is binary zero. The number of zeros in each field may vary; for example, 000.0.000 is equivalent to 0.0.0.

STACK_ALLOCATE_ Procedure

Summary

The STACK_ALLOCATE_ procedure allocates a user stack segment for use as a thread or alternate signal stack.

A user stack consists of three components in the following consecutive areas:

- a memory stack, growing downward from the highest-address end
- a guard area (one or more unmapped pages) that provides protection against overflow of the stack pointer
- a register stack (supporting the Register Stack Engine (RSE) of the Itanium processor), growing upward from the base (lowest address) of the segment

The size of each component, and therefore the overall segment, is a multiple of the memory page size of 16 KB. The caller can specify the minimum overall size of the stack segment, the minimum size of the guard area, allowance for growth, and the relative size of the RSE and memory areas.

If growth is enabled and the unmapped area exceeds the minimum size of the guard area, both the memory stack and the register stack areas can grow by mapping additional pages to them. Upon occurrence of a page fault for attempting to go beyond the currently mapped memory-stack area, that area is extended downward by consuming an available guard page. Upon occurrence of a page fault for attempting to go beyond the currently mapped RSE area, that area is extended upward by consuming an available guard page.

All pages within the stack described by *stackaddr* and *stacksize* have read and write permissions for the user.

The STACK_ALLOCATE_ procedure is a callable routine and is declared in kmem.h and KMEM.

Note. The STACK_ALLOCATE_ procedure is supported on systems running J06.10 and later J-series RVUs and H06.21 and later H-series RVUs. This procedure is not supported on G-series RVUs.

Syntax for C Programmers

```
#include <kmem.h>

short STACK_ALLOCATE_(
      unsigned int *stacksize                    /* i,o */
     ,unsigned int guardsize                     /* i   */
     ,void **stackaddr                           /*   o */
     ,unsigned int stackoptions);                /* i   */
```

Syntax for TAL Programmers

```
?SOURCE KMEM
error := STACK_ALLOCATE_(
      stacksize                                  ! i,o
     ,guardsize                                  ! i
     ,stackaddr                                  !   o
     ,stackoptions );                            ! i
```

- The STACK_ALLOCATE_ procedure is only supported in pTAL; it is not supported in TAL.

Parameters

error returned value

INT

indicates the outcome of the operation. Returns one of these values:

0 Operation was successful; *stackaddr* contains base address of the stack, and *stacksize* contains the rounded stack size.

1 Cannot allocate Kernel-Managed Swap space.

2 An invalid option value was specified.

3 A bounds violation on parameter.

5 Requested *stacksize* or *guardsize* is too large to handle; overall stack size limit is USERSTACK_MAX (16MB, defined in kmem.h).

15 Address space is unavailable.

If an error other than 0 is returned, the output parameter values are not set.

stacksize input, output

INT(32) .EXT:ref:1

specifies the overall stack size in bytes (which includes all the components of the stack) and returns the size actually allocated. The unsigned integer input value is

rounded up to a multiple of the memory page size (16 KB) and may be adjusted further as needed to satisfy the minimum size of each component. If any error is returned, *stacksize* is unchanged.

guardsize input

INT(32):value

specifies the minimum size of the guard area in bytes. The unsigned integer input value is rounded up to a multiple of the page size, with a minimum of one memory page. The size of the guard area is greater than its minimum when the stack has room for growth, because the guard area is the portion of the overall stack that has not been mapped.

stackaddr output

EXTADDR .EXT:ref:1

if the operation was successful, this output parameter contains the base address of the newly allocated stack and is always 16KB-page aligned. If any error other than 0 is returned, *stackaddr* is unchanged.

stackoptions input

INT(32):value

specifies special actions to be performed on the stack. This parameter is composed of multiple unsigned integer values, specifying attributes of the stack components and whether a special action is to be taken. Literals are defined in KMEM and kmem.h.

ST_NONE	specifies no bit set: all options default.
ST_COF	specifies that this stack segment is to be copied to a child process upon fork(), even if the stack is not currently active. By default, a thread stack is copied only if a thread running on this stack calls fork(). Stacks are not copied across exec().
ST_GROWTH**g*	where *g* is a constant in the range 0 to 100. This value specifies the percentage of the stack pages (excluding the minimum guard pages) to be reserved for growth (by remaining unmapped initially), subject to the constraint that the minimum size and granularity of each component is one page. The ST_GROWTH**g* percentage is applied to the total pages excluding the specified or default minimum guard pages; it is rounded down to whole pages (and can round to zero). The remaining pages are mapped as the segment is initialized, and have KMSF backing store reserved. When growth occurs, additional page(s) are mapped and backing store reserved. A value of 0 results in all pages (excluding the minimum number of guard pages) being initially mapped, with none reserved for growth. A value of 100 results in initially mapping just one page to the memory area and one page to the RSE area, with any remaining pages reserved for growth. A value

greater than 100 is erroneous; it will be rejected if less than 4096 but otherwise has undefined effect.

ST_RSE*r* — where *r* is a constant in the range 0 to 100. This value specifies the percentage of all initially mapped pages to be mapped to the register stack rather than the memory stack area. A value of 0 defaults to 50. A value greater than 100 is erroneous; it will be rejected if less than 4096 but otherwise has undefined effect. The ST_GROWTH**g* percentage is applied first; the pages not reserved for guard pages and growth are then apportioned to the memory and register stack areas. Any fractional page is rounded in favor of the memory stack area, subject to the constraint that each area has at least one page.

Examples

(1) Allocate one guard page, one memory-stack page, and one register-stack page:

```
stacksize = 0;
error = STACK_ALLOCATE_(&stacksize, 0, &stackaddr, ST_NONE);
```

Output stacksize is 48 KB (3 pages).

(2) Allocate one guard page, four memory-stack pages, and three register stack pages:

```
stacksize = 0x20000;   /* 128 KB */;
error = STACK_ALLOCATE_(&stacksize, 0, &stackaddr, ST_NONE);
```

Output stacksize is 128 K (8 pages).

(3) Allocate one guard page (minimum), four pages reserved for growth, three memory-stack pages, and two register-stack pages initially:

```
stacksize = 160000;
error = STACK_ALLOCATE_(&stacksize, 0, &stackaddr,
                        ST_GROWTH*50);
```

Output stacksize is 160 K (10 pages).

(4) Allocate two guard pages, two memory-stack pages, and one register-stack page initially:

```
stacksize = 75000;
error = STACK_ALLOCATE_(&stacksize, 20000, &stackaddr,
                        ST_NONE);
```

Output stacksize is 80 K (5 pages).

(5) Allocate one guard page, five memory-stack pages, and ten register-stack pages:

```
stacksize = 250000;
error = STACK_ALLOCATE_(&stacksize, 16384, &stackaddr,
                        ST_RSE*70);
```

Output stacksize is 256 K (16 pages).

(6) Allocate two guard pages (minimum), five pages reserved for growth, seven memory-stack pages, and two register-stack pages initially:

```
stacksize = 250000;
error = STACK_ALLOCATE_(&stacksize, 32000, &stackaddr,
                        ST_GROWTH*40 | ST_RSE*30);
```

Output stacksize is 256 K (16 pages).

STACK_DEALLOCATE_ Procedure

Summary

The STACK_DEALLOCATE_ procedure releases memory resources for a user stack that was allocated using the STACK_ALLOCATE_ procedure. The stack specified by the *stackaddr* parameter must have been allocated as a user stack in an earlier call to the STACK_ALLOCATE_ procedure or an error is returned.

If the specified stack is used as a signal stack, it must be de-registered prior to calling the STACK_DEALLOCATE_ procedure.

The STACK_DEALLOCATE_ procedure is a callable routine and is declared in kmem.h and KMEM.

Note. The STACK_DEALLOCATE_ procedure is supported on systems running J06.10 and later J-series RVUs and H06.21 and later H-series RVUs. This procedure is not supported on G-series RVUs.

Syntax for C Programmers

```
#include <kmem.h>

short STACK_DEALLOCATE_(
      void *stackaddr);                              /* i  */
```

Syntax for TAL Programmers

```
?SOURCE KMEM
error := STACK_DEALLOCATE_(stackaddr);                  ! i
```

- The STACK_DEALLOCATE_ procedure is only supported in pTAL; it is not supported in TAL.

Parameters

error returned value

INT

indicates the outcome of the operation. Returns one of these values:

0 Operation was successful.

2 The input parameter specified is not a valid user stack address.

4 Failed to deallocate user stack; either the stack is currently in use, or the stack is an active signal stack.

stackaddr input

EXTADDR:value

this input parameter specifies the base address of the user stack to be deallocated.

Example

```
error = STACK_DEALLOCATE_(stackaddr);
```

STEPMOM Procedure (Superseded by PROCESS_SETINFO_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The STEPMOM procedure is called by a process when it wants to receive process deletion (STOP or ABEND) messages for a process it did not create. Note that the caller of STEPMOM becomes the new “mom” of the designated process. (That is, STEPMOM replaces the mom field in the designated process’s process control block

extension with the 4-word process ID of its caller.) Therefore, only the caller receives the process deletion notification.

STEPMOM is typically used by the backup process of an unnamed process pair to monitor its primary process. (This monitoring is automatic between members of named process pairs.)

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
CALL STEPMOM ( process-id );                    ! i
```

Parameters

process-id input

INT:ref:4

is a 4-word array containing the process ID of an already executing process, for which the calling process wants to receive the process deletion message.

The process ID is a 4-word array, where:

[0:2]	Process name or creation timestamp
[3].<0:3>	Reserved
.<4:7>	Processor number where the process is executing
.<8:15>	PIN assigned by the operating system to identify the process in the processor

Condition Code Settings

< (CCL) indicates that STEPMOM failed, or that no process designated *process-id* exists.

= (CCE) indicates that the caller is now the creator (mom) of *process-id*.

> (CCG) is not returned from STEPMOM.

Considerations

- Process access ID and the caller of STEPMOM

  If STEPMOM is called from a Guardian process, the caller must either have the same process access ID as the process it is attempting to adopt, be the group manager of the process access ID, or be the super ID. For a description of the

process access ID, see Considerations on page 12-198 and to the *Guardian User's Guide*.

- OSS security

  If STEPMOM is called from an OSS process, the security rules that apply to calling STEPMOM are the same as those that apply to calling the OSS `kill()` function. For details, see the reference pages either online or in the *Open System Services System Calls Reference Manual*.

- Why STEPMOM should not be called for a process pair

  A process should not call STEPMOM for either member of a process pair. Adoption of a process pair by a third process causes errors and interferes with operation, because the operation depends upon each member of the process pair being the mom of the other.

- Adopting a single named process is not recommended

  If a single named process is adopted, the caller becomes both the mom and the ancestor and will receive two process termination messages when the process dies.

- STEPMOM and high-PIN processes

  You cannot use STEPMOM to adopt a high-PIN process because a high PIN cannot fit into *process-id*.

Figure 14-1 illustrates the effect of STEPMOM.

Messages

- Process deletion (STOP) message

  The caller of STEPMOM receives the process deletion (STOP) system message if the *process-id* is being deleted normally because of a call to STOP.

- Process deletion (ABEND) message

  The caller of STEPMOM receives the process deletion (ABEND) system message if the *process-id* is being deleted abnormally because of a call to ABEND, or because the process encountered a trap condition or received a process-terminating signal and is being deleted by the operating system.

OSS Considerations

If STEPMOM is used to set the mom of an OSS process, the new mom receives the Guardian process deletion message when the OSS process terminates. The message indicates that the terminated process was an OSS process and contains the OSS process ID; otherwise, the message is the same as one received for a terminating Guardian process.

If the OSS process successfully executes one of the OSS `exec` or `tdm_exec` set of functions, a Guardian process deletion message is sent to the mom. Although the process is still alive in the OSS environment (the OSS process ID still exists), the process handle no longer exists, so the process has terminated in the Guardian environment.

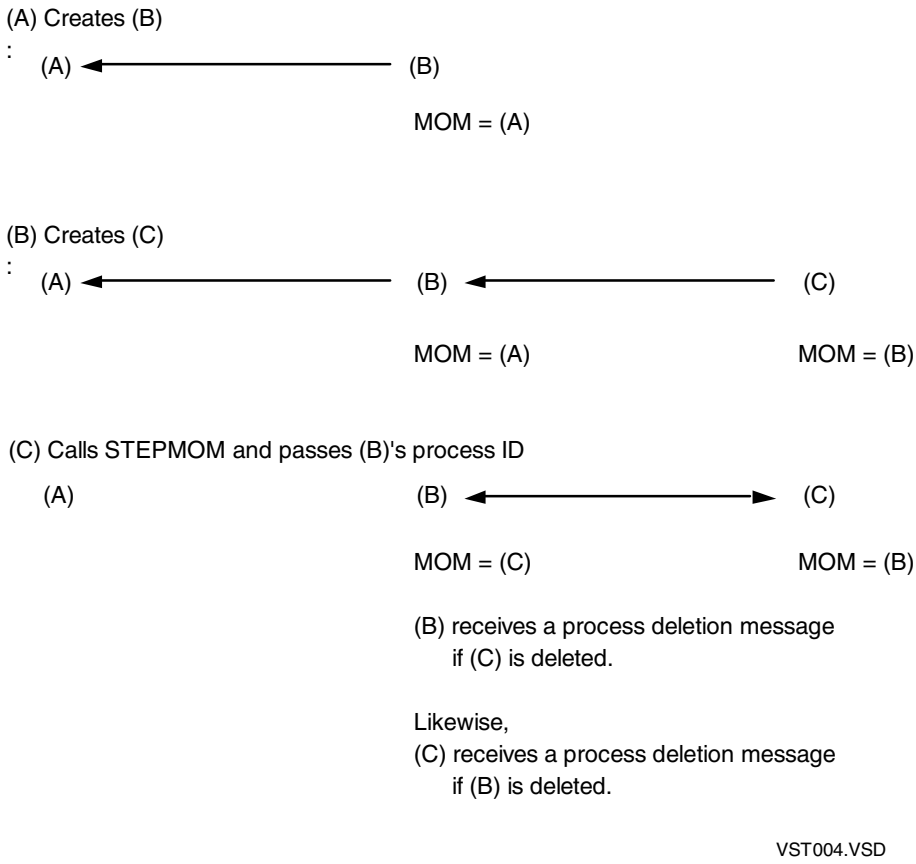
The OSS parent process (which is not necessarily the same process as the mom process) also receives OSS process termination status if the OSS process ID no longer exists. The order of delivery of the OSS process termination status and the Guardian process deletion message is not guaranteed.

For the format of the Guardian process deletion message, see the *Guardian Procedure Errors and Messages Manual*. For details on the OSS process termination status, see the `wait(2)` function reference page either online or in the *Open System Services System Calls Reference Manual*.

Example

```
CALL STEPMOM ( STEP^SON );
```

Figure 14-1. Effect of STEPMOM



STOP Procedure (Superseded by PROCESS_STOP_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The STOP procedure deletes a process or process pair and indicates that the deletion was caused by a normal condition. When this procedure is used to delete a Guardian process or an OSS process, a STOP system message is sent to the deleted process's creator. When this procedure is used to delete an OSS process, a `SIGCHLD` signal and the OSS process termination status are sent to the OSS parent.

STOP can be used by a process to:

- Delete itself
- Delete its own backup
- Delete another process

When the STOP procedure is used to delete a Guardian process, the caller must either have the same process access ID as the process it is attempting to stop, be the group manager of the process access ID, or be the super ID. For a description of the process access ID, see Considerations on page 12-198 and to the *Guardian User's Guide*.

When STOP is used on an OSS process, the same security rules apply as when using the OSS `kill()` function.

When STOP executes, all open files associated with the deleted process are automatically closed. If a process had BREAK enabled, BREAK is disabled.

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
CALL STOP ( [ process-id ]                                  ! i
          ,[ stop-backup ]                                  ! i
          ,[ error ]                                        ! o
          ,[ compl-code ]                                   ! i
          ,[ termination-info ]                             ! i
          ,[ spi-ssid ]                                     ! i
          ,[ length ]                                       ! i
          ,[ text ] );                                      ! i
```

Parameters

process-id input

INT:ref:4

indicates the process that is to be stopped. At this point, you have two options. The value you enter can be either:

- omitted (or zero), meaning “stop myself,” or
- the 4-word array containing the process ID of the process to be stopped, where:

[0:2]	Process name or creation timestamp
[3].<0:3>	Reserved
[3].<4:7>	Processor number where the process is executing
[3].<8:15>	PIN assigned by the operating system to identify the process in the processor

If *process-id*[0:2] references a process pair and *process-id*[3] is specified as -1, then both members of the process pair are stopped.

stop-backup input

INT:value

if specified as 1, the current process’s backup is stopped and STOP is returned to the caller. The *process-id* is not used.

If zero, this parameter is ignored, and the *process-id* parameter is used as described.

error output

INT:ref:1

returns a file-system error number. STOP returns a nonzero value for this parameter only when it cannot successfully make the request to stop the designated process. If it makes the request successfully (*error* is 0), the designated process might or might not be stopped depending on the stopmode of

the process and the authority of the caller. (The stopmode of the process can be changed; hence, a stop request that has inadequate authority to stop the process is saved by the system and might succeed at a later time.) See Considerations on page 14-163.

These parameters supply completion-code information which consists of four items: the completion code, a numeric field for additional termination information, a subsystem identifier in SPI format, and an ASCII text string. These items have meaning in the call to STOP only when a process is stopping itself.

compl-code input

INT:value

compl-code is the completion code to be returned to the creator process in the STOP system message and, for a terminating OSS process, in the OSS termination status. Specify this parameter only if the calling process is abending itself and you want to return a completion code value other than the default value of 0. For a list of completion codes, see Appendix C, Completion Codes.

termination-info input

INT:value

Can be provided as an option by the calling process if it is a subsystem process that defines Subsystem Programmatic Interface (SPI) error numbers. If supplied, this parameter should be the SPI error number that identifies the error that caused the process to stop itself. For more information on the SPI error numbers and subsystem IDs, see the *SPI Programming Manual*. If *termination-info* is not specified, this field is zero.

These parameters should be supplied to identify the subsystem ID that defines the SPI error number.

spi-ssid input

INT .EXT:ref:6

is a subsystem ID (SSID) that identifies the subsystem defining the *termination-info*. The format and use of the SSID is described in the *SPI Programming Manual*.

length input

INT:value

is the text length in bytes. Maximum is 80 bytes.

text input

STRING .EXT:ref:*length*

is an optional string of ASCII text to be sent in the STOP system message.

Condition Code Settings

A condition code value is returned only when a process is calling STOP on another process and that other process could not be stopped.

< (CCL) the *process-id* parameter is invalid, or an error occurred while stopping the process.

= (CCE) indicates that the STOP was successful.

> (CCG) does not return from STOP.

Considerations

- Differences between STOP and ABEND procedures

  When used to stop the calling process, the ABEND and STOP procedures operate almost identically; they differ in the system messages that are sent and the default completion codes that are reported. In addition, ABEND, but not STOP, causes a saveabend file to be created if the process's SAVEABEND attribute is set to ON. For information about saveabend files, see the *Inspect Manual*.

- Creator of the process and the caller of STOP

  If the caller of STOP is also the creator of the process being deleted, the caller receives the STOP system message.

- Rules for stopping a Guardian process: process access IDs and creator access IDs

  If the process is a local process and the request to stop it is also from a local process, these user IDs or associated processes may stop the process:

  - local super ID
  - the process's creator access ID (CAID) or the group manager of the CAID
  - the process's process access ID (PAID) or the group manager of the PAID

    If the process is a local process, a remote process cannot stop it.

    If the process is a remote process running on this node and the request to stop it is from a local process on this node, then these user IDs or associated processes may stop the process:

  - local super ID
  - the process's creator access ID (CAID) or the group manager of the CAID
  - the process's process access ID (PAID) or the group manager of the PAID

    If the process is a remote process on this node and the request to stop it is from a remote process, these user IDs or associated processes may stop the process:

  - a network super ID

- the process's network process access ID
- the process's network process access ID group manager
- the process's network creator access ID
- the process's network creator access ID group manager

  where network ID implies that the user IDs or associated process creators have matching remote passwords.

  Being local on a system means that the process has logged on by successfully calling VERIFYUSER on the system or that the process was created by a process that had done so. A process is also considered local if it is run from a program file that has the PROGID attribute set.

- Rules for stopping an OSS process

  The same rules apply when stopping an OSS process with the STOP procedure as apply for the OSS `kill()` function. See the `kill(2)` function reference page either online or in the *Open System Services System Calls Reference Manual*.

- Rules for stopping any process; stop mode

  When one process attempts to stop another process, another item checked is the stopmode of the process. Stopmode is a value associated with every process that determines which other processes can stop the process. The stopmode, set by procedure SETSTOP, is defined as follows:

  0 ANY other process can stop the process.
  1 ONLY the process qualified by the above rules can stop the process.
  2 NO other process can stop the process.

- Returning control to the caller before the process is stopped

  When *error* is 0, STOP returns control to the caller before the specified process is actually stopped. Although the process does not execute any more user code, you should make sure that it has terminated before you attempt to access a file that it had open with exclusive access or before you try to create a new process with the same name. The best way to be sure that a process has terminated is to wait for the process deletion message.

- Stopping a process that has the Inspect or saveabend attribute set

  If the process being stopped has either the Inspect attribute or the saveabend attribute set, and if DMON exists, STOP returns error 0 but deletion of the process is delayed until DMON approves it.

- Completion codes

  In response to the STOP procedure, the operating system supplies a completion code in the system message and, for OSS processes, in the OSS process termination status as follows:

- If a process calls STOP on another process, the system supplies a completion code value of 6.
- If a process calls STOP on itself but does not supply a completion code, the system supplies a completion code value of 0.

For a list of completion codes, see Appendix C, Completion Codes.

- Deleting high-PIN processes

  STOP cannot be used to delete a high-PIN unnamed process, but it can be used to delete a high-PIN *named* process or process pair.

  A high-PIN caller (named or unnamed) can delete itself by omitting *process-id*.

NetBatch Considerations

- The STOP procedure supports NetBatch processing by:
  - returning completion code information in the STOP system message
  - returning the process processor times in the STOP system message
  - sending a STOP system message to the ancestor of a job (GMOM) as well as the ancestor of a process

Messages

- Process deletion (STOP) message

  The creator of the stopped process is sent a system message -5 (process deletion: STOP), indicating that the deletion occurred. For the format of the interprocess system messages, see the *Guardian Procedure Errors and Messages Manual*

OSS Considerations

- When an OSS process is stopped by the STOP procedure, either by calling the procedure to stop itself or when some other process calls the procedure, the OSS parent process receives a `SIGCHLD` signal and the OSS process termination status. For details on the OSS process termination status, see the `wait(2)` function reference page either online or in the *Open System Services System Calls Reference Manual*.

  In addition, a STOP system message is sent to the mom, GMOM, or ancestor process according to the usual Guardian rules.

- When the STOP procedure is used to stop an OSS process other than the caller, the Guardian process ID must be specified in the call. The effect is the same as if the OSS `kill()` function was called with the input parameters as follows:
  - The *signal* parameter set to `SIGKILL`

- The *pid* parameter set to the OSS process ID of the process identified by *process-id* in the STOP call

- The security rules that apply to stopping an OSS process using STOP are the same as those that apply to the OSS `kill()` function. See the `kill(2)` function reference page either online or in the *Open System Services System Calls Reference Manual* for details.

Examples

```
CALL STOP;                  ! stop me.
CALL STOP ( ProcID );       ! stop the process that has
                            ! this process ID.
```

Related Programming Manual

For programming information on batch processing, see the *NetBatch User's Guide*.

STRING_UPSHIFT_ Procedure

Summary

The STRING_UPSHIFT_ procedure changes all the alphabetic characters in a string to upper case. Nonalphabetic characters remain unchanged.

Syntax for C Programmers

```
#include <cextdecs(STRING_UPSHIFT_)>

short STRING_UPSHIFT_ ( char *in-string
                       ,short length
                       ,char *out-string
                       ,short maxlen );
```

Syntax for TAL Programmers

```
error := STRING_UPSHIFT_ ( in-string:length          ! i:i
                          ,out-string:maxlen );      ! o:i
```

Parameters

error returned value

INT

indicates the outcome of the operation. Possible values are:

0	=	Operation successful
1	=	(reserved)
2	=	Parameter error
3	=	Bounds error
4	=	String too large to fit in *out-string*

in-string:*length* input:input

STRING .EXT:ref:*, INT:value

is the character string which is to have all alphabetic characters changed to upper case. *in-string* must be exactly *length* bytes long. The maximum acceptable value of *length* is 32,767.

out-string:*maxlen* output:input

STRING .EXT:ref:*, INT:value

returns the resultant string. The same buffer can be used for *in-string* and *out-string*.

maxlen is the length in bytes of the string variable *out-string*. *maxlen* must be at least as large as the length of the input string.

Example

```
err := STRING_UPSHIFT_ ( in^string:len, out^string:maxlen );
```

Related Programming Manual

For programming information about the STRING_UPSHIFT_ procedure, see the *Guardian Programmer's Guide*.

SUSPENDPROCESS Procedure (Superseded by PROCESS_SUSPEND_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The SUSPENDPROCESS procedure puts a process or process pair into the suspended state, preventing that process from being active (that is, executing instructions). (A process is removed from the suspended state and put back into the ready state if it is the object of a call to the ACTIVATEPROCESS procedure.)

Syntax for C Programmers

This procedure does not have a C syntax, because it is superseded and should not be used for new development. This procedure is supported only for compatibility with previous software.

Syntax for TAL Programmers

```
CALL SUSPENDPROCESS ( process-id );                    ! i
```

Parameters

process-id input

INT:ref:4

is a 4-word array containing the process ID of the process to be suspended, where:

[0:2]	Process name or creation timestamp
[3].<0:3>	Reserved
.<4:7>	Processor number where the process is executing
.<8:15>	PIN assigned by the operating system to identify the process in the processor

If *process-id*[0:2] references a process pair and *process-id*[3] is specified as -1, then both members of the process pair are suspended.

Condition Code Settings

< (CCL) indicates that SUSPENDPROCESS failed, or no process designated *process-id* exists.

= (CCE) indicates that *process-id* is suspended.

> (CCG) does not return from SUSPENDPROCESS.

Considerations

- When SUSPENDPROCESS is called on a Guardian process, the caller must either have the same process access ID as the process or process pair it is attempting to suspend, have super ID, or be the group manager of the process access ID. For information about system security, specifically the process access ID and super ID, see and the *Guardian User's Guide*.
- When SUSPENDPROCESS is called on an OSS process, the security rules that apply are the same as those that apply when calling the OSS `kill()` function. See the `kill(2)` function reference page either online or in the *Open System Services System Calls Reference Manual* for details.
- SUSPENDPROCESS cannot be used to suspend a high-PIN unnamed process. However, it can be used to suspend a high-PIN *named* process or process pair; *process-id*[3] must contain either -1 or two blanks.

  To suspend a high-PIN unnamed process, use the PROCESS_SUSPEND_ procedure. See the *Guardian Programmer's Guide*.

OSS Considerations

When used on an OSS process, SUSPENDPROCESS has the same effect as calling the OSS `kill()` function with the input parameters as follows:

- The *signal* parameter set to `SIGSTOP`
- The *pid* parameter set to the OSS process ID of the process identified by *process-id* in the SUSPENDPROCESS call

The `SIGSTOP` signal is delivered to the target process. The `SIGCHLD` signal is delivered to the parent of the target process.

Example

```
CALL SUSPENDPROCESS ( PROC^ID );     ! suspend process
```

SYSTEMENTRYPOINT_RISC_ Procedure

Summary

Note. pTAL syntax for this procedure is declared only in the EXTDECS0 file.

The SYSTEMENTRYPOINT_RISC_ procedure, which is defined only for native processes, returns either the 32-bit RISC address of the named entry point or, if not found, the value zero.

Syntax for C Programmers

```
#include <cextdecs(SYSTEMENTRYPOINT_RISC_)>

__int32_t SYSTEMENTRYPOINT_RISC_ ( char *name
                                  ,short len );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
risc-addr := SYSTEMENTRYPOINT_RISC_ ( name                ! i
                                     ,len );              ! i
```

Parameters

risc-addr returned value

EXTADDR

is the 32-bit RISC address.

name input

STRING .EXT:ref:*

is the case-sensitive entry point name.

len input

INT:value

is the length, in bytes, of *name*.

Example

```
EPNAME   ':=' "HEADROOM_ENSURE_";
RISC^ADDR :=  SYSTEMENTRYPOINT_RISC_ ( EPNAME , LEN );
```

SYSTEMENTRYPOINTLABEL Procedure

Summary

The SYSTEMENTRYPOINTLABEL procedure returns either the procedure label of the named entry point or, if not found, a zero.

Syntax for C Programmers

```
#include <cextdecs(SYSTEMENTRYPOINTLABEL)>

short SYSTEMENTRYPOINTLABEL ( char *name
                             ,short len );
```

Syntax for TAL Programmers

```
label := SYSTEMENTRYPOINTLABEL ( name              ! i
                                ,len );            ! i
```

Parameters

label returned value

INT

is the procedure label.

name input

STRING:ref:*

is the entry point name and must be specified in upper-case letters.

len input

INT:value

is the length, in bytes, of *name*.

Example

```
EPNAME ':=' "FILEINFO";
EPLABEL := SYSTEMENTRYPOINTLABEL ( EPNAME , LEN );
```

15 Guardian Procedure Calls (T-V)

This section contains detailed reference information for all user-accessible Guardian procedure calls beginning with the letters T through V. Table 15-1 lists all the procedures in this section.

Table 15-1. Procedures Beginning With the Letters T Through V

TAKE^BREAK Procedure

Summary

The TAKE^BREAK procedure enables BREAK monitoring for a file.

TAKE^BREAK is a sequential I/O (SIO) procedure and should be used only with files that have been opened by OPEN^FILE.

Syntax for C Programmers

```
#include <cextdecs(TAKE_BREAK)>

short TAKE_BREAK ( short _near *file-fcb );
```

Syntax for TAL Programmers

```
error := TAKE^BREAK ( file-fcb );          ! i
```

Parameters

error returned value

INT

is a file-system or sequential I/O (SIO) procedure error indicating the outcome of the operation.

file-fcb input

INT:ref:*

identifies the file for which BREAK is enabled. If the file is not a terminal, or if BREAK is already owned for this file, the call is ignored.

Considerations

- BREAK ownership and one terminal

Although the operating system allows a process to own BREAK on an arbitrary number of terminals, SIO supports BREAK ownership for only one terminal at a time.

- SIO does not support “break access”; SIO always issues SETMODE 11 with parameter 2 = 0.
- Taking BREAK ownership back

  If a process launches an offspring process that takes BREAK ownership, and the parent process then calls CHECK^BREAK, SIO takes BREAK ownership back. This can affect anticipated handling of BREAK.

Example

```
CALL TAKE^BREAK ( OUT^FILE );
```

Related Programming Manual

For programming information about the TAKE^BREAK procedure, see the *Guardian Programmer’s Guide*.

TEXTTOSSID Procedure

Summary

TEXTTOSSID scans a character string, expecting to find the external representation of a subsystem ID starting in the first byte (no leading spaces accepted). It returns the internal representation of the subsystem ID it finds.

Syntax for C Programmers

Note. In the TNS/E environment the CEXTDECS file uses the *int* data type for 32-bit values. This is a change from the TNS and TNS/R environments where CEXTDECS uses the *long* data type for 32-bit values.

```
#include <cextdecs(TEXTTOSSID)>

short TEXTTOSSID ( char *chars
                  ,short *ssid
                  ,[ __int32_t *status ] );
```

Syntax for TAL Programmers

```
len := TEXTTOSSID ( chars                ! i
                  ,ssid                  ! o
                  ,[ status ] );         ! o
```

Parameters

len returned value

INT

returns the number of characters scanned from *chars*.
Zero is returned if an error occurs.

chars input

STRING .EXT:ref:*

is the string containing the external representation of a subsystem ID. The number of characters scanned is returned as *len*. For the description of the external form of a subsystem ID, see “Considerations”.

ssid output

INT .EXT:ref:6

receives the internal form of the subsystem ID contained in *chars*.

status output

INT(32) .EXT:ref:1

a status code which indicates any problems encountered (the two numbers describe the two halves of the INT(32) value):

(0,0)no error

(0,x)something was wrong with the calling sequence

x: 29) a required parameter is missing
632not enough stack space

(1,x)error allocating the private segment; x is the error code from ALLOCATESEGMENT

(2,x)problem opening the template file.

x: > 0file-system error code
-1file code not 839 or 844
-2file not disk file
-3file not key sequenced

-4file has wrong record size
-5file has wrong primary key definition

(3,x)error reading the template file; x is the file-system error code.

(4,0)syntax error on the external form subsystem ID

(5,0)the subsystem ID had a subsystem name rather than a subsystem number and no match could be found for that name.

(6,0)the subsystem ID had a subsystem name rather than a subsystem number and more than one match was found for that name.

(7,x)error accessing the private segment; x is the error code returned from MOVEX.

(8,0)internal error

Considerations

The external form of a subsystem ID is:

owner.ss.version or 0.0.0

owner is 1 to 8 letters, digits, or hyphens, the first of which must be a letter; letters are not upshifted so the end user must enter *owner* in the proper case.

ss is either the subsystem number or the subsystem name.

A subsystem number is a string of digits which may be preceded by a minus sign. The value of the number must be between -32767 and 32767.

A subsystem name is 1 to 8 letters, digits, or hyphens, the first of which must be a letter. Letters are not upshifted; the end user must enter the subsystem name in the proper case.

version is either a string of digits which represents a TOSVERSION-format version (Ann) or a value from 0 to 65535.

Examples: HP.PATHWAY.C00
HP.52.0
0.0.0

The 0.0.0 form is used to represent the “null” subsystem ID. Its internal representation is binary zero. The number of zeros in each field may vary; for example, 000.0.000 is equivalent to 0.0.0.

TIME Procedure

Summary

The TIME procedure provides the current date and time in integer form.

Syntax for C Programmers

```
#include <cextdecs(TIME)>

void TIME ( short _near *date-and-time );
```

Syntax for TAL Programmers

```
CALL TIME ( date-and-time );                    ! o
```

Parameters

date-and-time output

INT:ref:7

returns an array with the current date and time in this form:

date-and-time[0]	year	(1983, 1984, ...)
[1]	month	(1-12)
[2]	day	(1-31)
[3]	hour	(0-23)
[4]	minute	(0-59)
[5]	second	(0-59)
[6]	.01 sec	(0-99)

Considerations

This procedure uses the 48-bit timestamp as the basis for determining the date and time. For a description of this form of timestamp, see TIMESTAMP Procedure.

Related Programming Manual

For programming information about the TIME utility procedure, see the *Guardian Programmer's Guide*.

TIMER_START_ Procedure (H-Series RVUs Only)

Summary

The TIMER_START_ procedure sets a timer to a given number of units of elapsed time, as measured by the processor clock. When the time expires, the calling process receives an indication in the form of a system message on $RECEIVE. TIMER_START_ measures the timeout value in microseconds. The process that calls this procedure becomes the owner of the underlying Time List element (TLE).

Syntax for C Programmers

```
error=TIMER_START_(timeoutValue, param1, param2, TLEid)
```

Syntax for TAL Programmers

```
INT(32) TIMER_START_ (TOV, PARAM1, PARAM2, TLEID)
                      CALLABLE, RESIDENT;
                      EXTERNAL;
```

Parameters

error return value

INT32

indicates the outcome of the call:

Value	Definition
0	Indicates the call is successful and the time has been started
590	Indicates the parameter values are invalid or inconsistent
3600	Indicates that TIMER_START_cannot allocate a TLE. This occurs when all available TLEs are used.

timeoutValue or TOV input

long long

a value greater than zero that specifies the time period (in microseconds) after which a timeout message should be queued on $RECIEVE. One second equals 1,000,000 microseconds.

param1 or PARAM1 input

UINT64

is part of the timeout message read from $RECEIVE

param2 or PARAM2 input

UINT64

is part of the timeout message read from $RECEIVE

TLEid or TLEID input

*INT32

the identifier associated with the timer. *TLEid* should only be used to call the TIMER_STOP_ procedure.

Considerations

- Param1 is delivered to the user process in the SIGNAL TIMEOUT message as an INT(16); it is truncated. Param2 is delivered in the SIGNALTIMEOUT message as an INT(32); it is also truncated.

Example

An example of how to set a 60 second timer is:

```
INT(32) TLEID;
ERR := TIMER_START_(60000000F, 10F, 32F, TLEID);
IF ERR THEN -- Timer was not started
```

TIMER_STOP_ Procedure (H-Series RVUs Only)

Summary

The TIMER_STOP_ procedure stops a timer started using the TIMER_START_ procedure. When called this procedure ensures that:

- the TLEid is valid
- the TLE owner is referred by the same TLEid
- the process that calls TIMER_STOP_ is the owner of the TLE

Syntax for C Programmers

```
error=TIMER_STOP_(TLEid)
```

Syntax for TAL Programmers

```
INT(32) TIMER_STOP_ (TLEID) CALLABLE, RESIDENT;
                    EXTERNAL;
```

Parameters

error return value

INT32

indicates the outcome of the call:

Value	Definition
0	Indicates the call is successful and the time has been stopped
<>0	Indicates the TLEid is invalid

TLEid or TLEID input

INT32

is the identifier associated with the timer. and returned by TIMER_START_.

TIMESTAMP Procedure

Summary

The TIMESTAMP procedure provides the internal form of the processor interval clock where the application is running.

Syntax for C Programmers

```
#include <cextdecs(TIMESTAMP)>

void TIMESTAMP ( short _near *interval-clock );
```

Syntax for TAL Programmers

```
CALL TIMESTAMP ( interval-clock );                ! o
```

Parameters

interval-clock output

INT:ref:3

returns the current value of the interval clock in a three-word array. A processor's interval clock is incremented every 0.01 second. The *interval-clock* parameter returns in this form:

[0]	most significant word, interval clock
[1]	interval clock
[2]	least significant word, interval clock

Considerations

- A 48-bit timestamp is a quantity equal to the number of 10-millisecond units since 00:00, 31 December 1974. The 48-bit timestamp always represents local civil time.

Procedures that work with the 48-bit timestamp are CONTIME, TIME, and TIMESTAMP.

- A 64-bit Julian timestamp is based on the Julian date. It is a quantity equal to the number of microseconds since January 1, 4713 B.C., 12:00 (noon) Greenwich mean time (Julian proleptic calendar). This timestamp can represent either Greenwich mean time, local standard time, or local civil time. There is no way to examine a Julian timestamp and determine which of the three times it represents.

  Procedures that work with the 64-bit Julian timestamp are COMPUTETIMESTAMP, CONVERTTIMESTAMP, INTERPRETTIMESTAMP, JULIANTIMESTAMP, and SETSYSTEMCLOCK.

- Process creation time is initialized by calling TIMESTAMP, which returns the local civil time in centiseconds (0.01 second = 10 milliseconds) since midnight (00:00) on 31 December 1974, in an array of three words. Only the two low-order words are saved in the process control block (PCB); this is sufficient to make the unnamed process ID unique.

- The RCLK instruction ($READCLOCK in TAL) is another source of timestamps. It returns a 64-bit timestamp representing the local civil time in microseconds since midnight (00:00) on 31 December 1974. Note that this is not a Julian timestamp.

- Process timing uses 64-bit elapsed time counters with microsecond resolution; these are not Julian timestamps either.

- There is no way to generalize about internal timing using 64-bit Julian timestamps or 48-bit timestamps. Each section of the operating system manages time using the method most appropriate for its application.

- All time and calendar units in this discussion are defined in *The Astronomical Almanac* published annually by the U.S. Naval Observatory and the Royal Greenwich Observatory.

Example

```
CALL TIMESTAMP ( TIMESTAMP^BUF );
```

Related Programming Manual

For programming information about the TIMESTAMP utility procedure, see the *Guardian Programmer's Guide*.

TOSVERSION Procedure

Summary

The TOSVERSION procedure identifies which version of the operating system is running.

Syntax for C Programmers

```
#include <cextdecs(TOSVERSION)>

short TOSVERSION ();
```

Syntax for TAL Programmers

```
version := TOSVERSION;
```

Parameter

version returned value

INT

returns a value of the form:

<0:7> an uppercase ASCII letter indicating the version of the operating system

<8:15> a binary number specifying the release number of the version

ASCII Letter	Operating-System Version
N	Dnn
P	Fnn
Q	Gnn
R	Hnn
T	Jnn

Example

```
VERSION := TOSVERSION;
```

For example, if the operating-system version is D10, the returned value contains “N” in bits <0:7> and binary 10 in bits <8:15>.

TS_NANOSECS_ Procedure (H-Series RVUs Only)

Summary

The TS_NANOSECS_ procedure returns a value that represents the time (in nanoseconds) since the last coldload. Because this procedure measures time in such fine detail, it may take a little longer to obtain timestamp information. The header files for this procedure can be found in cextdecs for C/C++ programs and in extdecs0 for pTAL programs.

Syntax for C Programmers

```
uint64 TS_NANOSECS_(void) ;
```

Syntax for TAL Programmers

```
FIXED PROC TS_NANOSECS_;
  EXTERNAL;
```

Consideration

- This procedure returns a value that represents time since cold-load in nanoseconds. As the resolution is in nanoseconds, the accuracy may be less depending upon the processor and the release of NonStop software.

TS_UNIQUE_COMPARE_ Procedure (H-Series RVUs Only)

Summary

The TS_UNIQUE_COMPARE_ procedure compares two unique timestamps created using TS_UNIQUE_CREATE_ Procedure (H-Series RVUs Only) and returns a value indicating their relationship. The header files for this procedure can be found in cextdecs for C/C++ programs and in extdecs0 for pTAL programs.

Syntax for C Programmers

```
TS_UNIQUE_COMPARE_ (short *TS1,
                    short *TS2);
```

Syntax for TAL Programmers

```
INT(32) PROC TS_UNIQUE_COMPARE_(TS1, TS2);
                                EXTERNAL;
```

Parameters

`TS1` `fixed`

is a 16-byte unique timestamp returned by TS_UNIQUE_CREATE_ Procedure (H-Series RVUs Only).

`TS2` `fixed`

is the second 16-byte unique timestamp returned by TS_UNIQUE_CREATE_ Procedure (H-Series RVUs Only)

TS_UNIQUE_COMPARE returns these values:

Return Value	Definition
TS_LESS_THAN [0]	The value of TS1 is less than the value of TS2. This value is returned only if TS1 and TS2 are created on the same CPU.
TS_IDENTICAL [1]	The value of TS1 is identical to the value of TS2. This value is returned only if TS1 and TS2 are created on the same CPU.
TS_GREATER_THAN [2]	The value of TS1 is greater than the value of TS2. This value is returned only if TS1 and TS2 are created on the same CPU.

Return Value	Definition
TS_AMBIGUOUS [3]	The values for TS1 and TS2 are nearly identical. This value is returned when TS1 and TS2 are created on the same system, but not the same CPU. If TS1 and TS2 are created on different systems, the difference in their creation time is less than one minute.
TS_PROBABLY_LESS_THAN [4]	The value of TS1 is probably less than the value of TS2. This value is returned when TS1 and TS2: ● Are created on the same system, but not the same CPU. The difference in their creation time is greater than one second. TS1 was probably created before TS2. -or- ● Are created on different systems. The difference in their creation time is probably more than one minute. TS1 was probably created before TS2.
TS_PROBABLY_GREATER_THAN [5]	The value of TS1 is probably greater than the value of TS2.This value is returned when TS1 and TS2: ● Are created on the same system, but not the same CPU. The difference in their creation time is greater than one second. TS1 was probably created before TS2. -or- ● Are created on different systems. The difference in their creation time is probably more than one minute. TS1 was probably created after TS2.

Consideration

- When timestamps are generated in different CPUs or on different systems in the same EXPAND network you can use the TS_UNIQUE_CONVERT_TO_JULIAN_ procedure to compare timestamps and obtain a close-to-exact result. TS_UNIQUE_CONVERT_TO_JULIAN_ allows you to extract the Julian timestamps from the Unique Timestamp and compare them using the normal compare operations. When using this method, the results are only as accurate as the time synchronization between the CPUs. This method does not allow you to determine which timestamp was created first when derived Julian timestamps are equal.

TS_UNIQUE_CONVERT_TO_JULIAN_ Procedure (H-Series RVUs Only)

Summary

The TS_UNIQUE_CONVERT_TO_JULIAN_ procedure converts a Unique Timestamp into a Julian timestamp. The header files for this procedure can be found in cextdecs for C/C++ programs and in extdecs0 for pTAL programs.

Syntax for C Programmers

```
TS_UNIQUE_CONVERT_TO_JULIAN_(short *TS,
                             uint64 * Julian);
```

Syntax for TAL Programmers

```
PROC TS_UNIQUE_CONVERT_TO_JULIAN_(TS, JULIAN);
             EXTERNAL;
```

Parameter

`TS` input

is a 16-byte unique timestamp returned by TS_UNIQUE_CREATE_ Procedure (H-Series RVUs Only). The timestamp is returned for use with the INTERPRETTIMESTAMP Procedure

`JULIAN` output

is a Julian timestamp.

TS_UNIQUE_CREATE_ Procedure (H-Series RVUs Only)

Summary

The TS_UNIQUE_CREATE_ procedure returns a 128-bit timestamp that is unique to the system it is generated on and any system in the same EXPAND network. The header files for this procedure can be found in cextdecs for C/C++ programs and in extdecs0 for pTAL programs.

The template definition of the unique 128-bit timestamp is:

```
STRUCT NSK_UNIQUETIMESTAMP128 (*);
 begin
    int(64) NS[0:1];
 end;
```

Note. The procedure calls, TS_UNIQUE_CONVERT_TO_JULIAN_ and TS_UNIQUE_COMPARE_, also use the NSK_UNIQUETIMESTAMP128 structure template.

Syntax for C Programmers

```
TS_UNIQUE_CREATE_(short *TS);
```

Syntax for TAL Programmers

```
INT PROC TS_UNIQUE_CREATE_(TS) RESIDENT CALLABLE;
                               EXTERNAL;
```

Parameters

`TS` output

is a unique timestamp returned.

TS_UNIQUE_CREATE returns these values:

Return Value	Definition
FEOK [0]	No error: The operation executed successfully.
FEBOUNDSERR [22]	One of the parameters specifies an address that is out of bounds.

Example

```
STRUCT TIMESTAMP(NSK_UniqueTimeStamp128);


ERR := TS_UNIQUE_CREATE_(TIMESTAMP);
IF ERR THEN
```

UNLOCKFILE Procedure

Summary

The UNLOCKFILE procedure unlocks a disk file and any records in that file currently locked by the user. The “user” is defined either as the opener of the file (identified by *filenum*) if the file is not audited—or the transaction (identified by the TRANSID) if the file is audited. Unlocking a file allows other processes to access the file. It has no effect on an audited file that has been modified by the current transaction.

Syntax for C Programmers

```
#include <cextdecs(UNLOCKFILE)>

_cc_status UNLOCKFILE ( short filenum
                      ,__int32_t tag );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.
- The function value returned by UNLOCKFILE, which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL UNLOCKFILE ( filenum                    ! i
                 ,[ tag ] );                 ! i
```

Parameters

filenum input

INT:value

is a number of an open file that identifies the file to be unlocked.

tag input

INT(32):value

is for nowait I/O only. *tag* is a value you define that uniquely identifies the operation associated with this UNLOCKFILE.

Note. The system stores the *tag* value until the I/O operation completes. The system then returns the *tag* information to the program in the *tag* parameter of the call to AWAITIO[X], thus indicating that the operation completed.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILE_GETINFO_ or FILEINFO).

= (CCE) indicates that the UNLOCKFILE was successful.

> (CCG) indicates that the file is not a disk file.

Considerations

- Nowait and UNLOCKFILE

  The UNLOCKFILE procedure must complete with a corresponding call to the AWAITIO[X] procedure when used with a file that is opened nowait.

- Locking queue

  If any users are queued in the locking queue for the file, the process at the head of the locking queue is granted access and is removed from the queue (the next read or lock request moves to the head of the queue).

- If the next user in the locking queue is waiting to:
  - lock the file or lock a record in the file, it is granted the lock (which excludes other users from accessing the file) and resumes processing.
  - read the file, its read is processed.

- Transaction Management Facility (TMF) and UNLOCKFILE

Locks on a file audited by TMF which has been modified by the current transaction are released only when the transaction is ended or aborted by TMF; in other words, a locked audited file which has been modified by the current transaction is unlocked during an ENDTRANSACTION or ABORTTRANSACTION processing for that file. An unmodified audited file is unlocked by UNLOCKFILE.

OSS Considerations

- This procedure operates only on Guardian objects. If an OSS file is specified, error 2 is returned

Example

```
CALL UNLOCKFILE ( SAVE^FILENUM );
```

Related Programming Manual

For programming information about the UNLOCKFILE file-system procedure, see the *Enscribe Programmer's Guide* and the *Guardian Programmer's Guide*.

UNLOCKREC Procedure

Summary

The UNLOCKREC procedure unlocks a record currently locked by the user. The "user" is defined either as the opener of the file (identified by *filenum*) if the file is not audited—or the transaction (identified by the TRANSID) if the file is audited. UNLOCKREC unlocks the record at the current position, allowing other users to access that record. UNLOCKREC has no effect on a record of an audited file if that record has been modified by the current transaction.

Syntax for C Programmers

```
#include <cextdecs(UNLOCKREC)>

_cc_status UNLOCKREC ( short filenum
                      ,__int32_t tag );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.
- The function value returned by UNLOCKREC, which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL UNLOCKREC ( filenum                    ! i
               ,[ tag ]  );                 ! i
```

Parameters

filenum input

INT:value

is the number of an open file that identifies the file containing the record to be unlocked.

tag input

INT(32):value

is for nowait I/O only. *tag* is a value you define that uniquely identifies the operation associated with this UNLOCKREC.

Note. The system stores this *tag* value until the I/O operation completes. The system then returns the *tag* information to the program in the *tag* parameter of the call to AWAITIO[X], thus indicating that the operation completed.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILE_GETINFO_ or FILEINFO).

= (CCE) indicates that the UNLOCKREC is successful.

> (CCG) indicates that the file is not a disk file.

Considerations

- File-opened nowait and UNLOCKREC

  The UNLOCKREC procedure must complete with a corresponding call to the AWAITIO[X] procedure when used with a file that is opened nowait.

- Queuing processes and UNLOCKREC

  If any users are queued in the locking queue for the record, the user at the head of the locking queue is granted access and is removed from the queue (the next read or lock request moves to the head of the queue).

  If the user granted access is waiting to lock the record, it is granted the lock (which excludes other process from accessing the record) and resumes processing.

  If the user granted access is waiting to read the record, its read is processed.

- Calling UNLOCKREC after KEYPOSITION

  If the call to UNLOCKREC immediately follows a call to KEYPOSITION where a nonunique alternate key is specified, the UNLOCKREC fails. A subsequent call to FILE_GETINFO_ or FILEINFO shows that error 46 (invalid key) occurred. However, if an intermediate call to READ or READLOCK is performed, the call to UNLOCKREC is permitted.

- Unlocking several records

  If several records need to be unlocked, the UNLOCKFILE procedure can be called to unlock all records currently locked by the user (rather than unlocking the records through individual calls to UNLOCKREC).

- Current-state indicators after UNLOCKREC

  For key-sequenced, relative, and entry-sequenced files, the current-state indicators after an UNLOCKREC remain unchanged.

- File pointers after UNLOCKREC

  For unstructured files, the current-record pointer and the next-record pointer remain unchanged.

- Transaction Management Facility (TMF) and UNLOCKREC

  A record that is locked in a file audited by TMF and has been modified by the current transaction is unlocked when an ABORTTRANSACTION or ENDTRANSACTION procedure is called for that file. Locks on modified records of audited files are released only when the transaction is ended or aborted by TMF. An unmodified audited record is unlocked by UNLOCKREC.

OSS Considerations

- This procedure operates only on Guardian objects. If an OSS file is specified, error 2 is returned

Example

```
CALL UNLOCKREC ( FILE^NUM );
```

Related Programming Manual

For programming information about the UNLOCKREC file-system procedure, see the *Enscribe Programmer's Guide* and the *Guardian Programmer's Guide*.

UNPACKEDIT Procedure

Summary

The UNPACKEDIT procedure converts a line image from EDIT packed line format into unpacked format. The input value is a text string in packed format, which includes

blank compression codes; the returned value is the same text in unpacked format, which can include sequences of blank characters.

UNPACKEDIT is an IOEdit procedure and is intended for use with files that have been opened by OPENEDIT or OPENEDIT_.

Syntax for C Programmers

```
#include <cextdecs(UNPACKEDIT)>

void UNPACKEDIT ( const char *packed-line
                 ,short packed-length
                 ,char *unpacked-line
                 ,short unpacked-limit
                 ,short *unpacked-length
                 ,[ short spacefill ]
                 ,[ short full-length ] );
```

Syntax for TAL Programmers

```
UNPACKEDIT ( packed-line                ! i
            ,packed-length              ! i
            ,unpacked-line              ! o
            ,unpacked-limit             ! i
            ,unpacked-length            ! o
            ,[ spacefill ]              ! i
            ,[ full-length ] );         ! i
```

Parameters

packed-line input

STRING .EXT:ref:*

is a string array that contains the line in packed format that is to be converted. The length of *packed-line* is specified by the *packed-length* parameter.

packed-length input

INT:value

specifies the length in bytes of *packed-line*. The *packed-length* must be in the range 1 through 256.

unpacked-line output

STRING .EXT:ref:*

is a string array that contains the line in unpacked format that is the outcome of the conversion. The length of the unpacked line is returned in the *unpacked-length* parameter.

unpacked-limit input

INT:value

specifies the length in bytes of the string variable *unpacked-line*.

unpacked-length output

INT .EXT:ref:1

returns the actual length in bytes of the value returned in *unpacked-line*. If *unpacked-line* is not large enough to contain the value that is the outcome of the conversion, *unpacked-length* returns a value of -1.

spacefill input

INT:value

if present and not equal to 0, specifies that if the value returned in *unpacked-line* is shorter than *unpacked-limit*, UNPACKEDIT should fill the unused part of *unpacked-line* with space characters. Otherwise, UNPACKEDIT does nothing to the unused part of *unpacked-line*.

full-length input

INT:value

if present and not equal to 0, specifies that all trailing space characters (if any) in the line being processed should be retained in the output line and should be counted in the value returned in *unpacked-length*. Otherwise, trailing space characters are discarded and not counted in *unpacked-length*.

Consideration

If it contains many blank characters, it is possible that the unpacked line might require much more memory than the packed line. To provide for this, you should specify a value for *unpacked-limit* that is at least fifteen times the value of than *packed-length*.

USER_AUTHENTICATE_ Procedure

Summary

The USER_AUTHENTICATE_ procedure verifies that a user exists and optionally logs on the user. This procedure should be called in a dialog mode to allow a dialog between the security mechanism and the application.

Syntax for C Programmers

```
#include <cextdecs(USER_AUTHENTICATE_)>

short USER_AUTHENTICATE_
                ( char *input text
                 ,short input text-len
                 ,[ short options ]
                 ,[ __int32_t *dialog-id ]
                 ,[ short *status ]
                 ,[ short *status-flags ]
                 ,[ char *displaytext ]
                 ,[ short displaytext-maxlen ]
                 ,[ short *displaytext-len ]
                 ,[ short cmon-timeout ]
                 ,[ char *termname ]
                 ,[ short termname-len ]
                 ,[ char *volsubvol ]
                 ,[ short volsubvol-maxlen ]
                 ,[ short *volsubvol-len ]
                 ,[ char *initdir ]
                 ,[ short initdir-maxlen ]
                 ,[ short *initdir-len ]
                 ,[ char *initprog ]
                 ,[ short initprog-maxlen ]
                 ,[ short *initprog-len ]
                 ,[ short *initprog-type ]
                 ,[ __int32_t *last-logon-time ]
                 ,[ __int32_t *time-password-expires ] );
```

- CEXTDECS (via the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
error := USER_AUTHENTICATE_
                  ( inputtext:inputtext-len              ! i:i
                   ,[ options ]                            ! i
                   ,[ dialog-id ]                          ! i,o
                   ,[ status ]                             ! o
                   ,[ status-flags ]                       ! o
                   ,[ displaytext:displaytext-maxlen ]     ! o:i
                   ,[ displaytext-len ]                    ! o
                   ,[ cmon-timeout ]                       ! i
                   ,[ termname:termname-len ]              ! i:i
                   ,[ volsubvol:volsubvol-maxlen ]         ! o:i
                   ,[ volsubvol-len ]                      ! o
                   ,[ initdir:initdir-maxlen ]             ! o:i
                   ,[ initdir-len ]                        ! o
                   ,[ initprog:initprog-maxlen ]           ! o:i
                   ,[ initprog-len ]                       ! o
                   ,[ initprog-type ]                      ! o
                   ,[ last-logon-time ]                    ! o
                   ,[ time-password-expires ] );           ! o
```

Parameters

error returned value

INT

returns an error number indicating the outcome of the call. Common errors returned are:

0 No error.

13 Invalid *termname* parameter.

22 Parameter out of bounds. An input parameter is not within the valid range, or return information does not fit into the length of the space provided, or an output parameter overlays the stack marker that was created by calling this procedure.

29 Missing parameter. Either this procedure is called without specifying input*text:*input*text-len*, or a parameter required by another parameter is not specified.

48 Security violation. The user specified in input*text:*input*text-len* is undefined, or an error occurred during a dialog with the Safeguard product. For detailed error information, see the *status* parameter.

70 Continue dialog. For detailed information on how to set the input*text* parameter in the next call to USER_AUTHENTICATE_, see the *status* parameter.

160 Invalid *dialog-id* parameter, invalid protocol, or dialog has exceeded two minutes.

563 The text to be returned in the *displaytext* parameter is longer than the length specified by the *displaytext-maxlen* parameter.

590 Two or more parameters provided are incompatible.

1554 Exceeded the maximum number of allowed concurrent requests.

For more information on file-system error messages, see the *Guardian Procedure Errors and Messages Manual*.

input*text*:input*text-len* input:input

STRING .EXT:ref:*, INT:value

specifies a user and password command string. If a dialog is not desired, input*text* must contain all the information necessary to complete the user authentication in a single call to the procedure.

input*text-len* specifies the length of the string variable input*text* in bytes. Only the first 256 characters of the string are used. This procedure does not return an error if the string exceeds 256 bytes in length.

For information on how to set input*text* to authenticate a user, log on, or change a password, see Considerations.

options input

INT:value

specifies additional requests of the USER_AUTHENTICATE_ procedure.

The bits, when set to 1, are defined as follows:

<0:1> Reserved (to be specified as 0).

<2> Enable PRIV-LOGON for users or aliases. It allows the programfile to logon as a user or alias when password is not supplied. No time delay is enforced on failure of USER_AUTHENTICATE_ to verify the user or alias with an invalid password.

<3> Reserved (specify 0).

<4> This bit is applicable only while setting and changing the long password (contains more than eight characters) using the USER_AUTHENTICATE_ procedure call. On all other events, this bit is ignored.

<5:6> Reserved (specify 0).

<7> Require password for all users (including the super ID).

<8> This bit must be set if USER_AUTHENTICATE_ is not being used in dialog mode (that is, only a single call to USER_AUTHENTICATE_ is being done) and either the blind logon bit is set or Safeguard is configured for blind logon. Otherwise, USER_AUTHENTICATE_ returns with *error* equal to 48 (security violation) and *status* equal to 4.

<9> Send $CMON the Prelogon^msg message (-59). This bit is valid either when Safeguard software is running and configured with CMON ON or when Safeguard software is not running.

<10> Send $CMON the Logon^msg message (-50). This bit is valid either when Safeguard software is running and configured with CMON ON or when Safeguard software is not running.

<11> Do not allow the super ID to log on.

<12> Do not allow a logon with a NonStop operating system user ID. When Safeguard software is running, the user can log on specifying either a member name or an alias. When Safeguard software is not running, the user can log on specifying a member name.

<13> Require blind logon. Setting this bit has the same effect as configuring BLINDLOGON using the Safeguard product. Passwords in input *text* are ignored unless bit 8 is set to assert that password echoing did not occur. Setting this bit is effective only on the first call of a dialog; it is ignored on the second or subsequent call.

If blind logon is set in bit 13 or configured using the Safeguard product, and a password is provided in input *text*, and bit 8 is set, authentication finishes in one call to USER_AUTHENTICATE_. If bit 8 is not set, then the outcome depends on *dialog-id*:

- If *dialog-ID* is supplied, then even if a password is provided in input *text*, USER_AUTHENTICATE_ returns with *error* equal to 70 and *status* equal to 4 to indicate that the password must be supplied in the next call to USER_AUTHENTICATE.
- If *dialog-id* is not supplied and a password is supplied in input *text*, then the USER_AUTHENTICATE_ returns with *error* equal to 48 and *status* equal to 4 to indicate a security violation.

<14> Do not log on if $CMON has an error or timeout. Setting this bit has the same effect as configuring Safeguard software with CMON ON and CMONERROR DENY. This bit has meaning only if CMON communication is attempted (either bit 9 or bit 10 is set, or the Safeguard software is configured with CMON ON).

<15> Log on and update the process's attributes to reflect the user's attributes. Following a successful logon with this procedure, the calling process is considered local with respect to the system on which it is running. Note that authentication occurs without logon when this bit is set to 0.

The default value is 0, which requests that the specified user be authenticated without logon, additional restrictions, or requests.

dialog-id input,output

FIXED .EXT:ref:1

specifies the identifier of the dialog and allows an authentication to take place over multiple calls to the procedure. To begin a dialog with USER_AUTHENTICATE_, set *dialog-id* to 0F. Use the *dialog-id* returned on each subsequent call to USER_AUTHENTICATE_ to continue the dialog. Error 70 (continue authentication dialog) is returned on each such call along with a status value that indicates the next required piece of information. The default value is 0F.

If *dialog-id* is not passed to USER_AUTHENTICATE_, then dialogs that require more than one call to USER_AUTHENTICATE_ are not possible. Error 48 (security violation) is returned instead of error 70. On return, *status* contains the same value as would have been returned with error 70.

status output

INT .EXT:ref:1

returns a value providing more information when *error* is 0, 48, or 70. *status* values are described in these three tables:

Values returned for *error* = 0 (no error):

status	**Description**
0	No status.
8	Password is valid but it is about to expire. Caller should return caution message.
22	The password is about to expire, and the caller has sent the new password (oldpassword, newpassword, newpassword). USER_AUTHENTICATE_ successfully changes the password to newpassword.
23	Long password is specified when the PASSWORD-COMPATIBILITY-MODE is set to ON. First eight characters of the specified password are accepted as the new password.

Values returned for *error* = 48 (security violation):

status	**Description**
1	User does not exist or password is incorrect.
2	Cannot authenticate with user ID because (1) *options* bit <12> is set to 1 and a Guardian user-id is passed, or (2) *options* bit <12> is set to 0 and a numeric Guardian user-id is passed and the SAFEGUARD NAMELOGON option is set to ON.
3	$CMON rejected the logon.
4	Conditions for a blind logon are not satisfied. This status bit is set when blind logon is configured (either through Safeguard software or using *options*.<13>), *dialog-id* is not provided, and *options*.<8> is not set.
5	Authentication record frozen.
6	Authentication record expired.
9	*dialog-id* was not provided. Caller should set *dialog-id* correctly and call USER_AUTHENTICATE_ again, with the same parameters.
10	*dialog-id* was not provided. Caller should set *dialog-id* correctly and call USER_AUTHENTICATE_ again, with the same parameters.
11	*dialog-id* was not provided. Caller should set *dialog-id* correctly and call USER_AUTHENTICATE_ again, with the same parameters.
12	*dialog-id* was not provided. Caller should set *dialog-id* correctly and call USER_AUTHENTICATE_ again, with the same parameters.
13	Password change request: new password is too short. New password is rejected.

status	Description
14	Password change request: new password is too long. New password is rejected.
15	Password change request: new password does not conform to password history (password cannot be reused). New password is rejected.
16	Password change request: new password does not conform to password quality. New password is rejected.
17	Password change request: new password contains blank characters. New password is rejected.
18	Password change request: new password is not verified. The first new password provided is not the same as the second new password provided.
19	Password change request: change cannot made during the allowed time period. Password change request is rejected.
20	Password change request: change is denied due to a system error. Retry later.
21	Cannot authenticate the super ID because *options* bit <11> is set to 1.
22	Cannot authenticate user because *options* bit <8> is not set, password is not present in the input *text* and *dialog-id* is not provided to USER_AUTHENTICATE_ when safeguard is not running.

Values returned for *error* = 70 (continue dialog):

status	Description
4	Caller should set input *text* to a password in the next call to USER_AUTHENTICATE_ to either log on or begin a password change.
9	Caller should set input *text* to a new password in the next call to USER_AUTHENTICATE_ to change the password.
11	Caller must set input *text* to a new password in the next call to USER_AUTHENTICATE_ to change the password, because the password has expired but the grace period is in effect. If the password is not changed, the user is not authenticated or logged on.
12	Caller should set input *text* to a new password in the next call to USER_AUTHENTICATE_ to verify the new password.

status-flags output

INT .EXT:ref:1

The bits, when set to 1, are defined as follows:

<0:14> Reserved.

<15> Caller should disable echo (with a SETMODE 20) for password input before the next call during this dialog with USER_AUTHENTICATE_.

displaytext:displaytext-maxlen output:input

STRING .EXT:ref:*, INT:value

if present and if *displaytext-maxlen* is not 0, returns a string, up to *displaytext-maxlen* bytes in length, representing one or more lines of logon dialog display text. Each line of text is preceded by a 16-bit length that is 16-bit aligned. The last line of text is followed by a byte length of 0 to indicate that there are no more display lines. Display text, generated by $CMON or Safeguard software, can be returned anytime during a dialog.

displaytext-maxlen is a value in the range 0 through 2048.

This parameter pair is required if *displaytext-len* is specified.

displaytext-len output

INT .EXT:ref:1

if *displaytext* is returned, contains its actual length in bytes.

This parameter is required if *displaytext:displaytext-maxlen* is specified.

cmon-timeout input

INT:value

specifies how many seconds the procedure should wait for $CMON to respond with pre-logon and logon messages. A value of 0 causes the procedure to wait indefinitely. The default value is 30 seconds. This parameter is ignored when Safeguard software is running.

termname:termname-len input:input

STRING .EXT:ref:*, INT:value

if supplied and if *termname-len* is not 0, is a file name that specifies the terminal for interaction with the security mechanism. If used, the value of *termname* must be exactly *termname-len* bytes long. If *termname* is partially qualified, it is resolved using the =_DEFAULTS DEFINE.

The default value is the home terminal of the caller.

volsubvol:volsubvol-maxlen output:input

STRING .EXT:ref:*, INT:value

if present and if *volsubvol-maxlen* is not 0, returns the name of the default volume and subvolume inherited by the specified user when a logon session is initiated. This information is returned in external form.

volsubvol-maxlen specifies the length of the string variable *volsubvol* in bytes.

This parameter pair is required if *volsubvol-len* is specified.

volsubvol-len output

INT .EXT:ref:1

if *volsubvol* is returned, contains its actual length in bytes.

This parameter is required if *volsubvol:volsubvol-maxlen* is specified.

initdir:initdir-maxlen output:input

STRING .EXT:ref:*, INT:value

if present and if *initdir-maxlen* is not 0, returns the OSS pathname for the initial working directory for the specified user in an OSS environment.

initdir-maxlen specifies the length of the string variable *initdir* in bytes.

This parameter pair is required if *initdir-len* is specified.

initdir-len output

INT .EXT:ref:1

if *initdir* is returned, contains its actual length in bytes.

This parameter is required if *initdir:initdir-maxlen* is specified.

initprog:initprog-maxlen output:input

STRING .EXT:ref:*, INT:value

if present and if *initprog-maxlen* is not 0, returns the OSS pathname for the initial program of the specified user in an OSS environment.

initprog-maxlen specifies the length of the string variable *initprog* in bytes.

You must either specify all of these parameters or specify none of them: *initprog:initprog-maxlen*, *initprog-len*, and *initprog-type*.

initprog-len output

INT .EXT:ref:1

if *initprog* is returned, contains its actual length in bytes.

You must either specify all of these parameters or specify none of them: *initprog:initprog-maxlen*, *initprog-len*, and *initprog-type*.

initprog-type output

INT .EXT:ref:1

returns 1 to indicate that the initial program type for the specified user in an OSS environment is an OSS program. Other values might be added in future RVUs.

You must either specify all of these parameters or specify none of them: *initprog:initprog-maxlen*, *initprog-len*, and *initprog-type*.

last-logon-time output

FIXED .EXT:ref:1

returns the Julian timestamp of when the specified user last logged on. If the specified user has never logged on, 0F is returned.

time-password-expires output

FIXED .EXT:ref:1

returns the Julian timestamp of when the password of the specified user expires. If either the password cannot be changed at the time USER_AUTHENTICATE_ is called or the password has no expiration date, 0F is returned.

Considerations

Note. An application should conduct a dialog with the security mechanism and determine the content of input *text* by the returned value of *status* when *error* is 70. The content of input *text* can change from RVU to RVU, so authentication in a single call to USER_AUTHENTICATE_ cannot be guaranteed.

- Conducting a dialog

  A dialog allows the application to interact with the security mechanism. To initiate a dialog, set *dialog-id* to 0F and set input *text* to user name. USER_AUTHENTICATE_ returns a new *dialog-id* to identify the next interaction with the procedure, returns *error* 70 to indicate a dialog, and a *status* value indicating the type of information that input *text* should have in the next call.

- Setting input *text* to authenticate a user and optionally log on

  To authenticate a user and to optionally log on, the call to USER_AUTHENTICATE_ must provide a user and usually a password. In this RVU, the user and password can be specified in input *text* as follows:

  ```
  "user, password"
  ```

  user is specified in input *text* by user name, user ID, or alias. A user ID cannot be specified when *options* bit <12> is set to 1. An alias cannot be specified when Safeguard software is not running.

  During a dialog, for example, input *text* can specify the user in the first call and the password in the second call.

- Setting input *text* to authenticate a user, log on, and change the password

  In this RVU, to log on a user and change a password, the call to USER_AUTHENTICATE_ must provide a user, a password, and two matching new passwords. A password can be changed only if Safeguard software is running. This information is specified in input *text* as follows:

  ```
  "user, password, newpassword, newpassword"
  ```

During a dialog, input *text* can specify the information in multiple calls. This example shows how input *text* could be set in three successive calls:

1. inputtext = “user”
2. inputtext = “password,”
3. inputtext = “newpassword, newpassword”

- Spaces in passwords

Blank spaces are not allowed in password. For example,

```
"  This is NOT a valid password"                    Invalid
"This    password   has too many embedded spaces"  Invalid
"This password is not valid "                       Invalid
```

- Authenticating a user

  If authentication without logon is requested (*options* bit <15> is 0), USER_AUTHENTICATE_ authenticates the user, but you cannot assume that user’s identity and you cannot log on. You must supply a password even if you do not request a logon unless:

  - You are the super ID (and *options*.<7> is not set to 1).
  - You are the group manager (*,255) (and *options*.<7> is not set to 1).
  - You are a user inquiring about yourself (and *options*.<7> is not set to 1).

- Logging on

  If authentication with logon is requested (*options* bit <15> is set to 1) and Safeguard software is running, and if the Safeguard parameter PASSWORD-REQUIRED is set to ON, you can assume that user’s ID if:

  - You know the user’s password.

  Alternatively, if authentication with logon is requested (*options* bit <15> is set to 1) and either Safeguard software is running, and the Safeguard parameter PASSWORD-REQUIRED is set to OFF or Safeguard software is not running, you can assume that user’s ID if:

  - You are the super ID (and *options*.<7> is not set to 1).
  - You are the group manager (*,255) (and *options*.<7> is not set to 1).
  - You know the user’s password.

  If any of these conditions are met, then your process access ID and creator access ID changes, you become a local user, and your default file security changes to what is established for the specified user.

- Disabling special authentication and logon privileges of the super ID and the group manager

If authentication is required regardless of who is executing the calling process, set *options*.<7> to 1. Setting this option overrides the special rules that otherwise allow the super ID or group manager to perform authentication or logon without providing the correct password. The effects of this option are enforced irrespective of whether Safeguard software is active and irrespective of whether *options*.<15> is set.

This bit enables server processes running as the super ID to check a requester's password without being able logon.

- Incorrect password timeout

  When Safeguard software is running, the number of times that a process can pass an invalid password to USER_AUTHENTICATE_ before the process is suspended and the length of time that the process is suspended are set during Safeguard configuration.

  When Safeguard software is not running, any process that passes an invalid password to USER_AUTHENTICATE_ for the third time is suspended for 60 seconds.

Following are the additional considerations:

- Setting the *options* bit 2 enables the privlogon functionality. This bit in conjunction with the value specified by bit 15 denotes the nature of privlogon requested. In systems where Safeguard is running a program file that invokes the USER_AUTHENTICATE_( ) with *options* bit 2 and 15 set to 1, and whose Safeguard disk-file attribute, PRIV-LOGON, is set to ON, may request for a successful logon without supplying a password. Similarly, a program file that invokes the USER_AUTHENTICATE_( ) with *options* bit 2 set to 1 and 15 set to 0, and whose Safeguard disk-file attribute, PRIV-LOGON, is set to ON, is not subjected to a time delay on supplying an incorrect password during authentication. The table below summarizes the functionalities:

Bit 2	Bit 15	Functionality
1	1	Privlogon (logon without passwords)
1	0	Privlogon (authenticate-only with no delay imposed on supplying incorrect password when Safeguard is up)
0	1	Regular logon (password necessary for logon)
0	0	Regular authenticate-only (time delay imposed on failure of successive password verification attempts)

- If a privlogon is requested through setting the *options* bit 2 and 15 to 1, when Safeguard software is running, and the PRIV-LOGON disk-file attribute of the programfile that invokes the USER_AUTHENTICATE_( ) is set to ON, the setting of bits 4, 7, 8, 9, 10, 12, 13 or 14 to ON is ignored.
- In systems where Safeguard is not running, a licensed program file that invokes USER_AUTHENTICATE_ () with *options* bit 2 and 15 set to 1, may request for a successful logon without supplying a password. However, a licensed program file invoking USER_AUTHENTICATE_ () with *options* bit 2 set to 1 and bit 15 set to 0, is subjected to a time delay on supplying a wrong password.

Note. This information is supported only on systems running H06.19 and later H-series RVUs.

- A password change request should not be specified during a privlogon request. The password change request is ignored and the password is not changed, when privlogon is requested through setting the *options* bit 2 and 15 to 1, when Safeguard software is running and the PRIV-LOGON disk-file attribute of the programfile that invokes the USER_AUTHENTICATE_( ) is set to ON.

Safeguard Considerations

If the Safeguard software has BLINDLOGON configured, USER_AUTHENTICATE_ enforces blind logon regardless of the setting of *options* bit <13> for blind logon.

OSS Considerations

- The *initdir* parameter indicates the OSS pathname for the initial working directory for the specified user in an OSS environment.
- The *initprog* parameter returns the OSS pathname for the initial program for the specified user in an OSS environment.

Example

This example shows how a user can be logged on and how to handle errors.

1. Set these parameters and call USER_AUTHENTICATE_:

   input*text* = user name
   options = bit <15> is set to 1
   dialog-id = 0F

2. If *error* = 70 and *status* = 4, then prompt the user for a password, disable echo with SETMODE 20, set these parameters, and call USER_AUTHENTICATE_:

   input*text* = password
   dialog-id = value of *dialog-id* returned from the previous call

3. Check *error* and *status* return values and enable echo with SETMODE 20.

- If *error* = 0 and *status* = 0, then the specified user is logged on.
- If *error* = 0 and *status* = 8, then the specified user is logged on but the password is about to expire. The caller displays a message telling the user to change the password by the *time-password-expires* value.
- If *error* = 0 and *status* = 22, then the specified user is logged on. The password was about to expire, and since the caller already passed in the new password along with the current password (password, newpassword, newpassword), the password was successfully changed to newpassword.
- If *error* = 0 and *status* = 23, then the specified password is long and the PASSWORD-COMPATIBILITY-MODE is set to ON. The first 8 characters of the specified password have been accepted as the new password. If the longer password contains the embedded blank spaces and invalid characters even after the 8th place, the password will be rejected though the PASSWORD-COMPATIBILITY-MODE is set to ON.
- If *error* = 70 and *status* = 11, then the specified user is not logged on because the password has expired. The caller displays a message telling the user that logon is denied due to password expiration. Another application could continue the dialog and prompt for a new password sequence.

Related Programming Manuals

For programming information on the command-interpreter monitor process ($CMON), see the *Guardian Programmer's Guide*. For more information on the Safeguard product, see the *Safeguard Reference Manual*.

USER_GETINFO_ Procedure

Summary

The USER_GETINFO_ procedure returns the default attributes of the specified user. The user can be identified by user name, by user ID, or if Safeguard software is running, by alias.

Syntax for C Programmers

Note. In the TNS/E environment the CEXTDECS file uses the *int* data type for 32-bit values. This is a change from the TNS and TNS/R environments where CEXTDECS uses the *long* data type for 32-bit values.

```
#include <cextdecs(USER_GETINFO_)>

short USER_GETINFO_
            ( [ char *user-name ]
             ,[ short user-maxlen ]
             ,[ short *user-curlen ]
             ,[ __int32_t *user-id ]
             ,[ short *is-alias ]
             ,[ short *group-count ]
             ,[ __int32_t *group-list ]
             ,[ __int32_t *primary-group ]
             ,[ char *volsubvol ]
             ,[ short volsubvol-maxlen ]
             ,[ short *volsubvol-len ]
             ,[ char *initdir ]
             ,[ short initdir-maxlen ]
             ,[ short *initdir-len ]
             ,[ char *initprog ]
             ,[ short initprog-maxlen ]
             ,[ short *initprog-len ]
             ,[ short *default-security ] )
             ,[ char *desc-field-text ]
             ,[ short desctxtfld-maxlen ]
             ,[ short *desctxt-len ]
             ,[ char *desc-field-bin ]
             ,[ short descbinfld-maxlen ]
              ,[ short *descbin-len ]);
```

Syntax for TAL Programmers

```
error := USER_GETINFO_
                  ( [ user-name:user-maxlen ]            !
i,o:i
                   ,[ user-curlen ]                      ! i,o
                   ,[ user-id ]                          ! i,o
                   ,[ is-alias ]                         ! o
                   ,[ group-count ]                      ! o
                   ,[ group-list ]                       ! o
                   ,[ primary-group ]                    ! o
                   ,[ volsubvol:volsubvol-maxlen ]       ! o:i
                   ,[ volsubvol-len ]                    ! o
                   ,[ initdir:initdir-maxlen ]           ! o:i
                   ,[ initdir-len ]                      ! o
                   ,[ initprog:initprog-maxlen ]         ! o:i
                   ,[ initprog-len ]                     ! o
                   ,[ default-security ] );              ! o
                   ,[ desc-field-text:desctxtfld-maxlen]!o:i
                   ,[ desctxt-len ]                      !o
                   ,[ desc-field-bin:descbinfld-maxlen] !o:i
                     ,[ descbin-len ])                      !o
```

Parameters

error returned value

INT

returns a file-system error number indicating the outcome of the call. Common errors returned are:

0	No error. Default user information is returned as requested.
11	Record not in file. The specified user name or user ID is undefined.
22	Parameter out of bounds. An input parameter is not within the valid range, or return information does not fit into the length of the space provided, or an output parameter supplied overlays the stack marker that was created by calling this procedure.
29	Missing parameter. This procedure was called without specifying a required parameter.
590	Bad parameter value. The value specified in *user-curlen* is greater than the value specified in *user-maxlen*, or the value specified in *user-curlen* is not within the valid range, or the value specified in *user-id* is not within the valid range.

For more information on file-system error messages, see the *Guardian Procedure Errors and Messages Manual*.

user-name:user-maxlen input, output:input

STRING .EXT:ref:*, INT:value

on input, if present and if *user-curlen* is not 0, specifies the user name or alias for which the default information is to be returned.

On output, if *user-id* is specified, this parameter returns the corresponding user name; otherwise, it remains unchanged from input.

user-name is passed, and returned, in one of two forms:

groupname.membername

The group name and member name are each up to 8 alphanumeric characters long, and the first character must be a letter. The group name and member name are separated by a period (.).

alias

The alias is a case-sensitive string made up of 1 through 32 alphanumeric characters, periods (.), hyphens (-), or underscores (_). The first character must be alphanumeric.

The *user-maxlen* parameter specifies the length of the string variable *user-name* in bytes.

This parameter pair is required if *user-curlen* is specified.

user-curlen input, output

INT .EXT:ref:1

on input, if *user-name* is specified, contains the actual length of *user-name* in bytes. A value of 0 indicates that *user-name* is treated as an output parameter and *user-id* is treated as an input parameter. The default value is 0.

On output, if *user-name* is returned, this parameter contains the actual length in bytes.

This parameter is required if *user-name:user-maxlen* is specified.

user-id input, output

INT(32) .EXT:ref:1

on input, if *user-curlen* is 0 or omitted, specifies the user ID for which the default information is to be returned.

On output, this parameter returns the user ID corresponding to the specified *user-name*.

The user ID is a value in the range 0 through 65,535, where the low-order two bytes, that is, the third and fourth bytes from the left, identify the group and the member respectively.

is-alias output

INT .EXT:ref:1

indicates whether a user name or an alias is supplied in *user-name*. This parameter returns these values:

-1 *user-name* is an alias.

0 *user-name* is a user name.

group-count output

INT .EXT:ref:1

returns the number of groups to which the specified user belongs. *group-count* is the number of valid elements in *group-list*.

group-list output

INT(32) .EXT:ref:32

returns a 32-element list containing the group IDs of the groups to which the specified user belongs. A user can belong to a minimum of one group, the primary group, and to a maximum of 32 groups. *group-list* contains *group-count* group IDs.

primary-group output

INT(32) .EXT:ref:1

returns the user's primary group ID.

volsubvol:volsubvol-maxlen output:input

STRING .EXT:ref:*, INT:value

if present and if *volsubvol-maxlen* is not 0, returns the name of the default volume and subvolume inherited by the specified user when a logon session is initiated. This information is returned in external form.

volsubvol-maxlen specifies the length of the string variable *volsubvol* in bytes.

This parameter pair is required if *volsubvol-len* is specified.

volsubvol-len output

INT .EXT:ref:1

if *volsubvol* is returned, contains its actual length in bytes.

This parameter is required if *volsubvol:volsubvol-maxlen* is specified.

initdir:initdir-maxlen output:input

STRING .EXT:ref:*, INT:value

if present and if *initdir-maxlen* is not 0, returns the OSS pathname for the initial working directory for the specified user in an OSS environment.

initdir-maxlen specifies the length of the string variable *initdir* in bytes.

This parameter pair is required if *initdir-len* is specified.

initdir-len output

INT .EXT:ref:1

if *initdir* is returned, contains its actual length in bytes.

This parameter is required if *initdir:initdir-maxlen* is specified.

initprog:initprog-maxlen output:input

STRING .EXT:ref:*, INT:value

if present and if *initprog-maxlen* is not 0, returns the OSS pathname for the initial program for the specified user in an OSS environment.

initprog-maxlen specifies the length of the string variable *initprog* in bytes.

This parameter pair is required if *initprog-len* is specified.

initprog-len output

INT .EXT:ref:1

if *initprog* is returned, contains its actual length in bytes.

This parameter is required if *initprog:initprog-maxlen* is specified.

default-security output

INT .EXT:ref:1

if present, returns the default file security to be used when a new file is created under the specified user ID. This information is returned in the form:

```
<0:3>    reserved
<4:6>    read
<7:9>    write
<10:12>  execute
<13:15>  purge
```

where the legitimate fields are encoded with numbers that represent this information:

0	A	(any local user)
1	G	(any local group member)
2	O	(only the local owner)

3		not used
4	N	(any network user)
5	C	(any network group/community user)
6	U	(only the network owner)
7	-	(only the local super ID)

These parameters are supported in G06.27 and all subsequent G-series RVUs.

desc-field-text:desctxtfld-maxlen output:input

STRING .EXT:ref:*, INT:value

If present and when *desctxtfld-maxlen* is not zero, returns the contents of the textual description field of the specified user/alias.

desctxt-len output

INT .EXT:ref:*

Returns the actual length of the specified user's textual description field. This parameter is required when *desc-field-text:desctxtfld-maxlen* is specified.

desc-field-bin:descbinfld-maxlen output:input

STRING .EXT:ref:*, INT:value

If present and when *descbinfld-maxlen* is not zero, returns the contents\of binary description field of the specified user/alias.

descbin-len output

INT .EXT:ref:*

Returns the actual length of the specified user's binary description field. This parameter is required, when *desc-field-bin:descbinfld-maxlen* is specified.

Considerations

- Either *user-id* or *user-name* must be supplied. If both parameters are supplied and *user-curlen* is greater than zero, then *user-name* is used and *user-id* is treated as an output parameter. In this case, no attention is paid to the current contents of the *user-id* parameter.
- For information on aliases or groups other than the primary group, or for OSS information, Safeguard software must be installed. For more information on the Safeguard product, see the *Safeguard Reference Manual*.

OSS Considerations

- The *initdir* parameter indicates the OSS pathname for the initial working directory for the specified user in an OSS environment.

- The *initprog* parameter returns the OSS pathname for the initial program for the specified user in an OSS environment.

Example

```
REALLEN := 0;
ERROR  := USER_GETINFO_ (USERNAME:MAXLEN, REAL^LEN, USER^ID);
          ! Get the username corresponding to a user ID
```

USER_GETNEXT_ Procedure

Summary

The USER_GETNEXT_ procedure returns the next user name or alias in the order in which it is stored by the security mechanism in effect. On successive calls, all user names and aliases can be obtained.

Syntax for C Programmers

```
#include <cextdecs(USER_GETNEXT_)>

short USER_GETNEXT_ ( [ char *user-name ]
                     ,[ short user-maxlen ]
                     ,[ short *user-curlen ]
                     ,[ short *is-alias ] );
```

Syntax for TAL Programmers

```
error := USER_GETNEXT_ ( user-name:user-maxlen   ! i,o:i
                        ,user-curlen             ! i,o
                        ,is-alias  );            ! i,o
```

Parameters

error returned value

INT

returns a file-system error number indicating the outcome of the call. Common errors returned are:

0 No error.

11 Record not in file. The specified *user-name* is undefined.

22 Parameter out of bounds. An input parameter is not within the valid range, or return information does not fit into the length of the space provided, or an output parameter overlays the stack marker that was created by calling this procedure.

29 Missing parameter. This procedure was called without specifying all parameters.

590 Bad parameter value. Either the value specified in *user-curlen* is greater than the value specified in *user-maxlen* or the value specified in *user-curlen* is not within the valid range.

For more information on file-system error messages, see the *Guardian Procedure Errors and Messages Manual* .

user-name:user-maxlen input, output:input

STRING .EXT:ref:*, INT:value

on input, specifies a character string that precedes the next *user-name* to be returned. *user-maxlen* specifies the length of the string variable *user-name* in bytes. To obtain the first user name or alias, set *user-curlen* to 0.

On output, this parameter returns the user name or alias that follows the *user-name* specified as the input parameter.

user-curlen input, output

INT .EXT:ref:1

on input, if *user-name* is specified, contains the actual length of *user-name* in bytes. To obtain the first user name or alias, set *user-curlen* to 0.

on output, returns the actual length of *user-name* in bytes.

is-alias input, output

INT .EXT:ref:1

on input, indicates whether the *user-name* input parameter contains a user name or an alias. This parameter can be set to these values:

0 *user-name* is a user name.

non-zero *user-name* is an alias.

On output, indicates whether the *user-name* output parameter contains a user name or an alias. This parameter returns these values:

-1 *user-name* is an alias.

0 *user-name* is a user name.

Considerations

- Aliases are defined only when Safeguard software is installed.
- This procedure returns a user name when *is-alias* is set to 0 on input and when *is-alias* returns 0 on output.
- This procedure returns an alias when *is-alias* is not set to 0.
- When the names are returned, user names are returned first and aliases are returned last. Furthermore, the return sequence is not circular: a user name never follows an alias. User names are not returned in any particular order, and the order can change from one RVU to the next. Similarly, aliases are not returned in any particular order and the order can change from one RVU to the next.

Example

```
!  put all user names on a system into name^list and
!  put all aliases on a system into alias^list
error := 0;
is^alias := 0;
name^entry := 0;
search^len := 0;
WHILE (error <> 0 and NOT is^alias) DO
   BEGIN
   error := USER_GETNEXT_
      (search^name:32, search^len, is^alias);
   name^list[name^entry].name ':='
      search^name FOR search^len BYTES;
   name^list[name^entry].len := search^len;
   name^entry := name^entry + 1;
   END;

IF is^alias THEN
   BEGIN
   alias^entry := 0;
   search^len := 0;
   ! decrement name^entry, because it was an alias
   name^entry := name^entry - 1;
   WHILE (error <> 0) DO
      BEGIN
      error := USER_GETNEXT_
         (search^name:32, search^len, is^alias);
      alias^list[alias^entry].alias ':='
         search^name FOR search^len BYTES;
      alias^list[alias^entry].len := search^len;
      alias^entry := alias^entry + 1;
      END;
   END;

IF error THEN
   BEGIN
   ! do error handling
   END
```

USERDEFAULTS Procedure (Superseded by USER_GETINFO_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development. This procedure does not support aliases or groups.

USERDEFAULTS returns the default attributes of the specified user, such as the user's default volume/subvolume and default file security. This same information is also available through VERIFYUSER; however, USERDEFAULTS is not restricted by security or password-validation considerations.

Syntax for C Programmers

USERDEFAULTS does not return condition codes when called from a C program.

Syntax for TAL Programmers

```
error := USERDEFAULTS ( [ user-id ]          ! i
                      ,[ user-name ]         ! i,o
                      ,[ volsubvol ]         ! o
                      ,[ filesecur ] );      ! o
```

Parameters

error returned value

INT

returns a file-system error number indicating the outcome of the call. Common errors returned are:

0	No error. Default user information is returned as requested.
1	End-of-file. The specified user ID or user name is undefined.
22	Parameter out of bounds. One of the parameters supplied overlays the stack marker that was created by calling this procedure.

29 Missing parameter. This procedure was called without specifying one of the required parameters (either *user-id* or *user-name*).

For more information on file-system error messages, see the *Guardian Procedure Errors and Messages Manual* .

user-id input

STRING .EXT:ref:2

specifies the user ID for which the default information is to be retrieved. Either *user-id* or *user-name* must be specified; if both are supplied, then *user-name* returns the user name corresponding to *user-id*. The user ID is passed in the form:

```
<0:7>    group ID    {0:255}
<8:15>   member ID {0:255}
```

user-name input, output

STRING .EXT:ref:16

specifies the user name for which the default information will be retrieved. Either *user-id* or *user-name* must be specified; if both are supplied, *user-name* returns the user name corresponding to *user-id*. The *user-name* is passed, and returned, in the form:

```
[0:3]    group name, blank-filled
[4:7]    member name, blank-filled
```

The group name and member name must both be input in uppercase.

volsubvol output

STRING .EXT:ref:16

if present, returns the name of the default volume and subvolume inherited by the specified user when a logon session is initiated. This information is returned in the form:

```
[0:3]    default volume, blank-filled
[4:7]    default subvolume, blank-filled
```

filesecur output

STRING .EXT:ref:2

if present, returns the default file security to be used when a new file is created under the specified user ID. This information is returned in the form:

```
<0:3>    reserved
<4:6>    read
<7:9>    write
```

<10:12> execute
<13:15> purge

where the legitimate fields are encoded with numbers that represent this information:

0	A	(any local user)
1	G	(any local group member)
2	O	(only the local owner)
3	not used	
4	N	(any network user)
5	C	(any network group/community user)
6	U	(only the network owner)
7	-	(only the local super ID)

Condition Code Settings

= (CCE) indicates that the default information is returned for the specified user.

> (CCG) indicates that the specified user ID or user name is undefined.

< (CCL) indicates that a required parameter is missing, that a buffer is out of bounds, or that an I/O error occurred on the user ID file ($SYSTEM.SYSTEM.USERID).

Considerations

Either *user-id* or *user-name* must be supplied. When both parameters are supplied, then *user-id* is used and *user-name* is treated as an output parameter; in this case, no attention is paid to the current contents of the *user-name* parameter.

Example

Suppose that a process needs to acquire default user information about its actual creator (process access id). Code similar to this example might be used to obtain such information:

```
INT   ERROR;
INT   USERID;
INT  .USERNAME[ 0:7 ];
INT   USERFILESEC;
INT  .USERVOLSVOL[ 0:7 ];
.
.
.
USERID := CREATORACCESSID;
ERROR  := USERDEFAULTS (USERID,      USERNAME,
                        USERVOLSVOL, USERFILESEC);
IF ERROR  THEN
   BEGIN  ! error !
   .
   .
   END
ELSE
```

```
BEGIN  ! successful !
.
.
END;
```

USERIDTOUSERNAME Procedure (Superseded by USER_GETINFO_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development. This procedure does not support aliases or groups.

The USERIDTOUSERNAME procedure returns the user name, from the file $SYSTEM.SYSTEM.USERID, that is associated with a designated user ID.

Syntax for C Programmers

```
#include <cextdecs(USERIDTOUSERNAME)>

_cc_status USERIDTOUSERNAME ( short _near *id-name );
```

- The function value returned by USERIDTOUSERNAME, which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL USERIDTOUSERNAME ( id-name );              ! i,o
```

Parameters

id-name input, output

INT:ref:8

on input, contains the user ID to be converted to a user name. The user ID is passed in the form:

id-name.<0:7>	group ID	{0:255}
.<8:15>	member ID	{0:255}

on the return, contains the user name associated with the specified user ID in the form:

id-name	FOR 4	group name, blank-filled
id-name[4]	FOR 4	member name, blank-filled

Condition Code Settings

< (CCL) indicates that *id-name* is out of bounds or that an I/O error occurred with the $SYSTEM.SYSTEM.USERID file.

= (CCE) indicates that the designated user name returned.

> (CCG) indicates that the specified user ID is undefined.

USERIOBUFFER_ALLOW_ Procedure

Summary

The USERIOBUFFER_ALLOW_ procedure dynamically sets the process behavior to be the same as if the objectfile flag "**allow_user_buffers**" in eld, provided it is called before opening any files. Its impact is limited to files opened by FILE_OPEN_ after the procedure is called. It has no impact on I/O operations initiated on files opened before the call or files opened by OPEN.

Syntax for C Programmers

```
void USERIOBUFFER_ALLOW_(void)
```

Syntax for TAL Programmers

```
PROC USERIOBUFFER_ALLOW_   ;
```

For NSAA systems, the default is system buffers for I/O operation on all files. Calling the USERIOBUFFER_ALLOW_ procedure enables user buffers for I/O operations (for example READX or WRITEX) on all subsequent files opened by FILE_OPEN_ procedure.
The USERIOBUFFER_ALLOW_ procedure does not affect the buffer status of files opened by the OPEN procedure, which uses system buffers for its I/O buffers. SETMODE 72 can enable user buffers on the files that are already opened.

The USERIOBUFFER_ALLOW_ procedure is useful on NSAA systems and can be called on any system (H-series or J-series). Once the procedure is called, the USERIOBUFFER_ALLOW_ setting cannot be turned off.

The user buffers being enabled does not guarantee that the user buffers will be used; the system is still free to select the most efficient buffers to use. In practice, I/O less than 4096 bytes will system buffers.

The user buffers should be in multiples of page size and page aligned for optimal performance. They must have at least 8-byte alignment.

When using user buffers, read or write operations must not be performed on nowait user buffers until the AWAITIO procedure is called.

USERNAMETOUSERID Procedure (Superseded by USER_GETINFO_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development. This procedure does not support aliases or groups.

The USERNAMETOUSERID procedure returns the user ID, from the file $SYSTEM.SYSTEM.USERID, that is associated with a designated user name.

Syntax for C Programmers

```
#include <cextdecs(USERNAMETOUSERID)>

_cc_status USERNAMETOUSERID ( short _near *name-id );
```

- The function value returned by USERNAMETOUSERID, which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL USERNAMETOUSERID ( name-id );        ! i,o
```

Parameters

name-id input, output

INT:ref:8

on input, contains the user name to be converted to a user ID. The user name is passed in the form:

name-id	FOR 4	group name, blank-filled
name-id[4]	FOR 4	member name, blank-filled

The group name and member name must both be input in uppercase.

on the return, contains the user ID associated with the specified user name in the form:

name-id.<0:7>	group ID	{0:255}
name-id.<8:15>	member ID	{0:255}

Condition Code Settings

< (CCL) indicates that *name-id* is out of bounds or that an I/O error occurred with the $SYSTEM.SYSTEM.USERID file.

= (CCE) indicates that the designated user ID returned.

> (CCG) indicates that the specified user name is undefined.

USESEGMENT Procedure (Superseded by SEGMENT_USE_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development.

The USESEGMENT procedure selects a particular extended data segment to be currently addressable by the calling process.

For selectable segments, a call to USESEGMENT must follow a call to ALLOCATESEGMENT to make the selectable extended data segment accessible. Although you can allocate multiple selectable extended data segments, you can access only one at a time.

For shared flat segments, a call to USESEGMENT can follow a call to ALLOCATESEGMENT, but calling USESEGMENT is unnecessary because all of the flat segments allocated by a process are always accessible to the process.

Flat segments and selectable segments are supported on native processors that use D30 or later versions of the HP NonStop operating system. Selectable segments are supported on all systems.

Syntax for C Programmers

- You cannot call USESEGMENT directly from a C program, because it returns a value and also sets the condition-code register. To access this procedure, you must write a “jacket” procedure in TAL that is directly callable by your C program. For information on how to do this, see the discussion of procedures that return a value and a condition code in the *C/C++ Programmer’s Guide*. Note that the SEGMENT_USE_ procedure, which should be used in new development, does not require a “jacket” procedure in TAL to be called from a C program.

Syntax for TAL Programmers

```
old-segment-id := USESEGMENT [ ( segment-id ) ];      ! i
```

Parameters

old-segment-id returned value

INT

returns the segment ID of the previously used selectable segment, if any; otherwise, it returns -1.

If *segment-id* specifies a flat segment, *old-segment-id* returns the segment ID of the current in-use selectable segment. The flat segment and the selectable segment remain addressable by the calling process.

segment-id input

INT:value

if present, is the segment ID of the segment is to be used or -1 if no segment is used. If this parameter is not supplied, the current in-use selectable segment remains unchanged and *old-segment-id* returns the current in-use selectable segment ID. Note that *old-segment-id* is returned even if *segment-id* is not valid, in which case the *old-segment-id* parameter continues to remain the in-use segment.

Condition Code Settings

< (CCL) indicates that *segment-id* is not allocated, or that the segment cannot be used by a nonprivileged caller.

= (CCE) indicates that the operation is successful.

> (CCG) does not return from USESEGMENT.

Considerations

- Because segment relocation is done, the first byte of any selectable extended data segment has the address %2000000D (%H00080000).
- Selectable segments and performance

  If you have more than one selectable segment, you might face performance degradation, because time is wasted when switching between the selectable segments. This is because only one selectable segment is visible at a time. Instead, use flat segments, which are always visible.
- See Considerations on page 2-25.

Example

```
INT seg^id1 := 0;
INT seg^id2 := 1;
INT old^seg^id;
INT status;
INT(32) seg^len := %177777D;        ! 64K - 1 bytes

status := ALLOCATESEGMENT ( seg^id1, seg^len );
IF status <> 0 THEN ...
status := ALLOCATESEGMENT ( seg^id2, seg^len );
IF status <> 0 THEN ...

old^seg^id := USESEGMENT ( seg^id1 );     ! use first segment
IF <> THEN ...

old^seg^id := USESEGMENT ( seg^id2 );  ! change segments
IF <> THEN ...
```

VRO_SET_ Procedure (H-Series RVUs Only)

Summary

VRO_SET_ causes a voluntary rendezvous opportunity to take place.

Syntax for C Programmers

```
void VRO_SET_(void);
```

Syntax for TAL Programmers

```
proc VRO_SET_;
```

VERIFYUSER Procedure (Superseded by USER_AUTHENTICATE_ Procedure and USER_GETINFO_ Procedure)

Summary

Note. This procedure is supported for compatibility with previous software and should not be used for new development. This procedure does not support aliases or groups.

VERIFYUSER has three different functions. It is used to control logons, to verify that a user exists, and to return user-default information. Logon control and user verification is also available through the USER_AUTHENTICATE_ procedure. User-default information is also available through USER_GETINFO_ without the security restrictions and password validation required by VERIFYUSER.

Syntax for C Programmers

```
#include <cextdecs(VERIFYUSER)>

_cc_status VERIFYUSER ( short _near *user-name-or-id
                       ,[ short logon ]
                       ,[ short _near *default ]
                       ,short default-len );
```

- The function value returned by VERIFYUSER, which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL VERIFYUSER (user-name-or-id                    ! i
                ,[ logon ]                          ! i
                ,[ default                          ! o
                ,[ default-len ] );                 ! i
```

Parameters

user-name-or-id input

INT:ref:12

is an array containing the name or user ID of the user to be verified or logged on, as follows:

```
[0:3]     group name, blank-filled
[4:7]     member name, blank-filled
```

The group name and member name must both be input in uppercase.

or

```
[0].<0:7>  group ID
[0].<8:15> member ID
[1:7]      zeros (ASCII nulls)
```

In either case:

```
[8:11]     password, blank-filled (see "Considerations")
```

logon input

INT:value

if present, has this meaning:

```
   0  verify user but do not log on
<> 0  verify user and log on
```

if omitted, is assumed to have a value of 0.

default output

INT:ref:18

if present, returns information regarding the user specified in *user-name-or-id*:

```
[0:3]          group name, blank-filled
[4:7]          member name, blank-filled
[8].<0:7>      group ID
   .<8:15>     member ID
[9:12]         default volume, blank-filled
[13:16]        default subvolume, blank-filled
[17].<0:15>    default file security, as follows:

               <0:3>    reserved
               <4:6>    read
               <7:9>    write
               <10:12>  execute
               <13:15>  purge
```

where the legitimate fields are encoded with numbers that represent this information:

0	A	(any local user)
1	G	(any local group member)
2	O	(only the local owner)
3	not used	
4	N	(any network user)
5	C	(any network group/community user)
6	U	(only the network owner)
7	-	(only the local super ID)

default-len input

INT:value

is the length, in bytes, of the *default* array. *default-len* is required if *default* is specified. This number should always be specified as 36; in the future, new fields can be added to *default*, requiring *default-len* to become larger.

Condition Code Settings

< (CCL) indicates that a buffer is out of bounds, or that an I/O error occurred on the user ID file ($SYSTEM.SYSTEM.USERID).

= (CCE) indicates a successful verification or logon.

> (CCG) indicates that there is no such user or that the password is invalid.

Considerations

Condition code CCE returns under these conditions:

- Specifying 0 for the *logon* parameter verifies that there is a user with that name on the system, but you cannot assume that user's identity and you cannot log on. You must supply a password even if you specify 0 for the *logon* parameter, unless:
  - You are the super ID.
  - You are the group manager (*,255).
  - You are a user inquiring about yourself.
- If the *logon* parameter is a value other than 0 and the Safeguard parameter PASSWORD-REQUIRED is set to OFF, you can assume that user's ID if:
  - You are the super ID.
  - You are the group manager (*,255).
  - You know the user's password.

  If you assume one of the above IDs, then your process access ID and creator access ID changes, you become a local user, and your default file security changes to what is established in the local USERID file.

- Following a successful logon with this procedure, the calling process is considered local with respect to the system on which it is running.
- A process that passes an invalid password to VERIFYUSER for the third time is suspended for 60 seconds.
- Note that each call to VERIFYUSER always results in an open, KEYPOSITION, READUPDATEUNLOCK, WRITEUPDATEUNLOCK, and close operation on the USERID file.
- System users are defined through the TACL ADDUSER command. All TACL commands are described in the *TACL Reference Manual*.

Example

```
USER := 3 '<' 8 + 17;                  ! user ID 3,17.
USER[1] ':=' 0 & USER[1] FOR 6;        ! all zeros.
USER[8] ':=' password FOR 8;
LOGON := 1;                            ! log this user on.

CALL VERIFYUSER ( USER , LOGON , DEFAULT , DEFAULT^LEN );

IF < THEN ...                          ! buffer or I/O error,
ELSE IF > THEN ...                     ! no such user, or bad
                                       ! password.
ELSE ...                               ! successful.      .
```

The array “USER” is prepared with the member and group ID and then passed to VERIFYUSER. VERIFYUSER logs on the process with the member ID 17 and group ID 3.

16

Guardian Procedure Calls (W-Z)

This section contains detailed reference information for all user-accessible Guardian procedure calls beginning with the letters W through Z. Table 16-1 lists all the procedures in this section.

Table 16-1. Procedures Beginning With the Letters W Through Z

WAIT^FILE Procedure

Summary

The WAIT^FILE procedure is used to wait or check for the completion of an outstanding I/O operation.

WAIT^FILE is a sequential I/O (SIO) procedure and should be used only with files that have been opened by OPEN^FILE.

Syntax for C Programmers

```
#include <cextdecs(WAIT_FILE)>

short WAIT_FILE ( short _near *file-fcb
                 ,[ short _near *count-read ]
                 ,[ __int32_t time-limit ] );
```

- CEXTDECS (through the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
error := WAIT^FILE ( file-fcb                  ! i
                   ,[ count-read ]             ! o
                   ,[ time-limit ] );          ! i
```

Parameters

error returned value

INT

If abort-on-error mode is in effect, the only possible values for *error* are:

0	No error
1	End of file
6	System message (only if user requested system messages through SET^SYSTEMMESSAGES or SET^SYSTEMMESSAGESMANY)
40	Operation timed out (only if *time-limit* is supplied and is not -1D)
111	Operation aborts because of BREAK (if BREAK is enabled)
532	Operation restarted

file-fcb input

INT:ref:*

identifies the file for which there is an outstanding I/O operation.

count-read output

INT:ref*

if present, is the count of the number of bytes returned due to the requested read operation. The value returned to the parameter has no meaning when waiting for a write operation to complete.

time-limit input

INT(32):value

if present, indicates whether the caller waits for completion or checks for completion. If omitted, the time limit is set to -1D.

time-limit <> 0D	indicates a wait for completion. The time limit then specifies the maximum time, in 0.01-second units, the caller waits for a completion.
0D	indicates a check for completion. WAIT^FILE immediately returns to the caller regardless of whether

there is a completion. If no completion occurs the I/O operation is still outstanding; an *error* 40 and an "operation timed out" message are returned.

0D (and *error* = 40) There is no completion. Therefore, READ^FILE or WRITE^FILE cannot be called for the file until the operation completes by WAIT^FILE.

-1D indicates a willingness to wait forever.

Example

```
ERROR := WAIT^FILE ( IN^FILE , COUNT );
```

Related Programming Manual

For programming information about the WAIT^FILE procedure, see the *Guardian Programmer's Guide*.

WRITE[X] Procedures

Summary

The WRITE[X] procedures write data from an array in the application program to an open file (see "Considerations").

The WRITE procedure is intended for use with 16-bit addresses, while WRITEX is intended for use with 32-bit extended addresses. Therefore, the data buffer for WRITEX can be either in the caller's stack segment or any extended data segment.

Syntax for C Programmers

```
#include <cextdecs(WRITE)>

_cc_status WRITE ( short filenum
                  ,short _near *buffer
                  ,unsigned short write-count
                  ,[unsigned short _near *count-written ]
                  ,[ __int32_t tag ] );


#include <cextdecs(WRITEX)>

_cc_status WRITEX ( short filenum
                   ,const char _far *buffer
                   ,unsigned short write-count
                   ,[unsigned short _far *count-written ]
                   ,[ __int32_t tag ] );
```

- CEXTDECS (through the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.
- The function value returned by WRITE[X], which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL WRITE[X] ( filenum                          ! i
               ,buffer                           ! i
               ,write-count                      ! i
               ,[ count-written ]                ! o
               ,[ tag ]  );                      ! i
```

Parameters

filenum input

INT:value (Use with WRITE and WRITEX)

is the number of an open file that identifies the file to be written.

buffer input

INT:ref:* (Use with WRITE)
STRING .EXT:ref:* (Use with WRITEX)

is an array containing the information to be written to the file.

write-count input

INT:value (Use with WRITE and WRITEX)

is the number of bytes to be written:

{0:57344}	for disk files (see Disk File Considerations on page 16-8 and Appendix J, System Limits)
{0:32755}	for terminal files
{0:57344}	for other nondisk files (device-dependent)
{0:57344}	for interprocess files
{0:80}	for the operator console

For key-sequenced and relative files, 0 is invalid. For entry-sequenced files, 0 indicates an empty record.

count-written output

INT:ref:1 (Use with WRITE)
INT .EXT:ref:1 (Use with WRITEX)

is for wait I/O only. *count-written* returns a count of the number of bytes written to the file.

tag input

INT(32):value (Use with WRITE and WRITEX)

is for nowait I/O only. *tag* is a value you define that uniquely identifies the operation associated with this WRITE[X].

Note. The system stores this *tag* value until the I/O operation completes. The system then returns the *tag* information to the program in the *tag* parameter of the call to AWAITIO[X], thus indicating that the operation completed. If WRITEX is used, you must call AWAITIOX to complete the I/O. If WRITE is used, you can use either AWAITIO or AWAITIOX to complete the I/O.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILE_GETINFO_ or FILEINFO).

< (CCL) is returned following successful insertion or update of a record in a file with one or more insertion-ordered alternate keys if a duplicate key value was created for at least one insertion-ordered alternate key. A call to FILE_GETINFO_ or FILEINFO shows that error 551 occurred; this error is advisory only and does not indicate an unsuccessful write operation.

= (CCE) indicates that the WRITE[X] is successful.

> (CCG) indicates that an error occurred (call FILE_GETINFO_ or FILEINFO).

Considerations

- WRITE versus WRITEX

  Use WRITE when the buffer has a 16-bit address, and use WRITEX when the buffer has a 32-bit extended address. Therefore, the data buffer for WRITEX can be either in the caller's stack segment or any extended data segment.

- Waited I/O and WRITE[X]

  If a waited WRITE[X] is executed, the *count-written* parameter indicates the number of bytes actually written.

- Nowait I/O and WRITE[X]

  If a nowait WRITE[X] is executed, *count-written* has no meaning and can be omitted. The count of the number of bytes written is obtained when the I/O operation completes through the *count-transferred* parameter of the AWAITIO[X] procedure.

  The WRITE[X] procedure must complete with a corresponding call to the AWAITIO[X] procedure when used with a file that is opened nowait. If WRITEX is used, you must call AWAITIOX to complete the I/O. If WRITE is used, you can use

either AWAITIO or AWAITIOX to complete the I/O.

▲ **WARNING.** When using nowait file I/O, data corruption might occur if the WRITE buffer is modified before the AWAITIOX that completes the call.

You should not change the contents of the data buffer between the initiation and completion of a nowait write operation. This is because a retry can copy the data again from the user buffer and cause the wrong data to be written. You should avoid sharing a buffer between a write and another I/O operation because this creates the possibility of changing the contents of the write buffer before the write is completed.

Disk File Considerations

- Large data transfers for unstructured files using default mode

  For the write procedures (WRITE[LOCK] [UNLOCK]), default mode allows I/O sizes for unstructured files to be as large as 56 KB (57,344), excepting writes to audited files, if the unstructured buffer size (or block size) is 4 KB (4096). Default mode here refers to the mode of the file if SETMODE function 141 is not invoked.

  For an unstructured file with an unstructured buffer size other than 4 KB, DP2 automatically adjusts the unstructured buffer size to 4 KB, if possible, when an I/O larger than 4KB is attempted. However, this adjustment is not possible for files that have extents with an odd number of pages; in such cases an I/O over 4 KB is not possible. Note that the switch to a different unstructured buffer size will have a transient performance impact, so it is recommended that the size be initially set to 4 KB, which is the default. Transfer sizes over 4 KB are not supported in default mode for unstructured access to structured files.

- Large data transfers using SETMODE 141

  For WRITEX only, large data transfers (more than 4096 bytes) can be done for files opened with unstructured access, regardless of unstructured buffer size, by using SETMODE function 141. When SETMODE 141 is used to enable large data transfers, it is permitted to specify up to 56K (57344) bytes for the *write-count* parameter. For use of SETMODE function 141, see Table 14-4 on page 14-63.

- File is locked

  If a call to WRITE[X] is made and the file is locked through a file number other than that supplied in the call, the call is rejected with file-system error 73 (file is locked).

- Inserting a new record into a file

  The WRITE[X] procedure inserts a new record into a file in the position designated by the file's primary key:

Key-Sequenced Files	The record is inserted in the position indicated by the value in its primary-key field.

Queue Files	The record is inserted into a file at a unique location. The disk process sets the timestamp field in the key, which causes the record to be positioned after the other existing records that have the same high-order user key. If the file is audited, the record is available for read operations when the transaction associated with the write operation commits. If the transaction aborts, the record is never available to read operations. If the file is not audited, the record is available as soon as the write operation finishes successfully. Unlike other key-sequenced files, a write operation to a queue file will never encounters an error 10 (duplicate record) because all queue file records have unique keys generated for them.
Relative Files	After an open or an explicit positioning by its primary key, the record is inserted in the designated position. Subsequent WRITE[X]s without intermediate positioning insert records in successive record positions. If -2D is specified in a preceding positioning, the record is inserted in an available record position in the file. If -1D is specified in a preceding positioning, the record is inserted following the last position used in the file. There does not have to be an existing record in that position at the time of the WRITE[X].

Note. If the insert is to be made to a key-sequenced or relative file and the record already exists, the WRITE[X] fails, and a subsequent call to FILE_GETINFO_ or FILEINFO shows that error 10 occurred.

Entry-Sequenced Files	The record is inserted following the last record currently existing in the file.
Unstructured Files	The record is inserted at the position indicated by the current value of the next-record pointer.

- Structured files
  - Inserting records into relative and entry-sequenced files

    If the insertion is to a relative or entry-sequenced file, the file must be positioned currently through its primary key. Otherwise, the WRITE[X] fails, and a subsequent call to FILE_GETINFO_ or FILEINFO shows that error 46 (invalid key) occurred.
  - Current-state indicators after WRITE[X]

After a successful WRITE[X], the current-state indicators for positioning mode and comparison length remain unchanged.

For key-sequenced files, the current position and the current primary-key value remain unchanged.

For relative and entry-sequenced files, the current position is that of the record just inserted and the current primary-key value is set to the value of the record's primary key.

- Duplicate record found on insertion request

  When attempting to insert a record into a key-sequenced file, if a duplicate record is found, the WRITE[X] procedure returns error 10 (record already exists) or error 71 (duplicate record). If the operation is part of a TMF transaction, the record is locked for the duration of the transaction.

- Unstructured files
  - DP2 BUFFERSIZE rules

    DP2 unstructured files are transparently blocked using one of the four valid DP2 blocksizes (512, 1024, 2048, or 4096 bytes; 4096 is the default). This transparent blocksize, known as BUFFERSIZE, is the transfer size used against an unstructured file. While BUFFERSIZE does not change the maximum unstructured transfer (4096 bytes), multiple I/Os may be performed to satisfy a user request depending on the BUFFERSIZE chosen. For example, if BUFFERSIZE is 512 bytes, and a request is made to write 4096 bytes, at least eight transfers, each 512 bytes long, will be made. More than eight transfers happen, in this case, if the requested transfer does not start on a BUFFERSIZE boundary.

    DP2 performance with unstructured files is best when requested transfers begin on BUFFERSIZE boundaries and are integral multiples of BUFFERSIZE.

  - If the WRITE[X] is to an unstructured disk file, data is transferred to the record location specified by the next-record pointer. The next-record pointer is updated to point to the record following the record written.

  - The number of bytes written

    If an unstructured file is created with the odd unstructured attribute (also known as ODDUNSTR) set, the number of bytes written is exactly the number specified in *write-count*. If the odd unstructured attribute is not set when

the file is created, the value of *write-count* is rounded up to an even value before the WRITE[X] is executed.

You set the odd unstructured attribute with the FILE_CREATE_, FILE_CREATELIST_, or CREATE procedure, or with the File Utility Program (FUP) SET and CREATE commands.

- File pointers after WRITE[X]

  After a successful WRITE[X] to an unstructured file, the file pointers have these values:

  current-record pointer := next-record pointer; next-record pointer := next-record pointer + *count written*; end-of-file (EOF) pointer := max (EOF pointer, next-record pointer);

Interprocess Communication Consideration

- Indication that the destination process is running

  If the WRITE[X] is to another process, successful completion of the WRITE[X] (or AWAITIO[X] if nowait) indicates that the destination process is running.

Considerations for WRITEX Only

- The buffer and count transferred may be in the user stack or in an extended data segment. The buffer and count transferred cannot be in the user code space.
- If the buffer or count transferred is in a selectable extended data segment, the segment must be in use at the time of the call. Flat segments allocated by a process are always accessible to the process.
- If the file is opened for nowait I/O, the extended segment containing the buffer need not be in use at the time of the call to AWAITIOX.
- If the file is opened for nowait I/O, and the buffer is in an extended data segment, you cannot deallocate or reduce the size of the extended data segment before the I/O completes with a call to AWAITIOX or is canceled by a call to CANCEL or CANCELREQ.
- If the file is opened for nowait I/O, you must not modify the buffer before the I/O completes with a call to AWAITIOX. This also applies to other processes that may be sharing the segment. It is the application's responsibility to ensure this.
- If the file is opened for nowait I/O, and the I/O has been initiated with these routines, the I/O must be completed with a call to AWAITIOX (not AWAITIO).
- Nowait I/O initiated with these routines may be canceled with a call to CANCEL or CANCELREQ. The I/O is canceled if the file is closed before the I/O completes or

AWAITIOX is called with a positive time limit and specific file number and the request times out.

- If the extended address of the buffer is odd, bounds checking rounds the address to the next lower word boundary and checks an extra byte as well. The odd address is used for the transfer.

Errors for WRITEX Only

In addition to the errors returned from the WRITE procedure, file-system error 22 is returned when:

- The address of a parameter refers to the selectable segment area but no selectable segment is in use at the time of the call.
- The address of a parameter is extended, but it is an absolute address and the caller is not privileged.
- The file system cannot use the user's segment when needed.

Example

```
CALL WRITE ( OUT^FILE , OUT^BUFFER , 72 );
```

Related Programming Manuals

For programming information about the WRITE file-system procedure, see the *Guardian Programmer's Guide*, the *Enscribe Programmer's Guide*, and the data communication manuals.

WRITE^FILE Procedure

Summary

The WRITE^FILE procedure writes a file sequentially. The file must be open with write or read/write access.

WRITE^FILE is a sequential I/O (SIO) procedure and should be used only with files that have been opened by OPEN^FILE.

Syntax for C Programmers

```
#include <cextdecs(WRITE_FILE)>

short WRITE_FILE ( short _near *file-fcb
                  ,short _near *buffer
                  ,short write-count
                  ,[ short reply-error-code ]
                  ,[ short forms-control-code ]
                  ,[ short nowait ] );
```

Syntax for TAL Programmers

```
error := WRITE^FILE ( file-fcb                    ! i
                     ,buffer                      ! i
                     ,write-count                 ! i
                     ,[ reply-error-code ]        ! i
                     ,[ forms-control-code ]      ! i
                     ,[ nowait ] );               ! i
```

Parameters

error returnedvalue

INT

is a file-system or sequential I/O (SIO) error indicating the outcome of the write.

If abort-on-error mode, the default case, is in effect, the only possible values for *error* are:

0 No error

111 Operation aborted because of BREAK (if BREAK is enabled)

If *nowait* is not 0, the only possible value for *error* is 0, when abort-on-error mode is in effect.

file-fcb input

INT:ref:*

identifies the file to which data is written.

buffer input

INT:ref:*

is the data to be written. *buffer* must be located within ‘G’[0:32767], the process data area.

write-count input

INT:value

is the count of the number of bytes of *buffer* to be written. A *write-count* value of -1 causes SIO to flush the block buffer associated with the *file-fcb* passed.

reply-error-code input

INT:value

(for $RECEIVE file only) if present, is a file-system error to return to the requesting process by REPLY. If omitted, 0 is returned.

forms-control-code input

INT:value

(optional) indicates a forms-control operation to be performed before executing the actual WRITE when the file is a process or a line printer. The *forms-control* paramater corresponds to *parameter* of the file-system CONTROL procedure for *operation* equal to 1. No forms control is performed if *forms-control* is omitted, if it is -1, or if the file is not a process or a line printer.

nowait input

INT:value

if present, indicates whether to wait in this call for the I/O to complete. If omitted or zero, then wait is indicated. If *nowait* is not zero, the I/O must be completed in a call to WAIT^FILE.

Example

```
CALL WRITE^FILE ( OUT^FILE, BUFFER, COUNT );
```

Related Programming Manual

For programming information about the WRITE^FILE procedure, see the *Guardian Programmer's Guide*.

WRITEEDIT Procedure

Summary

The WRITEEDIT procedure accepts a line in unpacked format, converts it into EDIT packed line format, and writes it to the specified file.

WRITEEDIT is an IOEdit procedure and can only be used with files that have been opened by OPENEDIT or OPENEDIT_.

Syntax for C Programmers

Note. In the TNS/E environment, the CEXTDECS file uses the *int* data type for 32-bit values. This is a change from the TNS and TNS/R environments where CEXTDECS uses the *long* data type for 32-bit values.

```
#include <cextdecs(WRITEEDIT)>

short WRITEEDIT ( short filenum
                 ,[ __int32_t record-number ]
                 ,char *unpacked-line
                 ,short unpacked-length
                 ,[ short full-length ]
                 ,[ __int32_t *new-record-number ] );
```

Syntax for TAL Programmers

```
error := WRITEEDIT ( filenum                          ! i
                    ,[ record-number ]                ! i
                    ,unpacked-line                    ! i
                    ,unpacked-length                  ! i
                    ,[ full-length ]                  ! i
                    ,[ new-record-number ] );         ! o
```

Parameters

error returnedvalue

INT

returns a file-system error number indicating the outcome of the operation. Possible values include:

10 File already includes a line with the specified record number.
21 Specified record is too long to fit into EDIT packed line format. (The maximum is 255 bytes of packed text.)
45 All of the file's possible extents are allocated and full. You can use EXTENDEDIT to increase the file's extent size and call WRITEEDIT again.

filenum input

INT:value

specifies the file number of the open file to which the line is to be written.

record-number input

INT(32):value

if present, specifies the record number of the line to be written. If *record-number*:

- is greater than or equal to 0, it specifies 1000 times the EDIT line number of the line to be written.
- is -1, the line is written at the beginning of the file.
- is -2, the line is written at the end of the file.
- is -3, the line is written to the line represented by the file's current record number.

If this parameter is omitted, -3 is used.

unpacked-line input

STRING .EXT:ref:*

is a string array that contains the line in unpacked format that is to be written. The length of *unpacked-line* is specified by the *unpacked-length* parameter.

unpacked-length input

INT:value

specifies the length in bytes of *unpacked-line*. The minimum value is 0 bytes and the maximum value is the equivalent of 255 bytes of packed text. The maximum value of *unpacked-length* is variable because the packing algorithm depends on the number of sequences of blank characters.

full-length input

INT:value

if present and not equal to 0, specifies that all trailing space characters (if any) in the line being processed should be retained in the output line image. Otherwise, trailing space characters are discarded.

new-record-number output

INT(32) .EXT:ref:1

returns the record number of the newly written line. This value is 1000 times the EDIT line number of the line.

Example

```
INT(32) record^num := -2D;  ! write to end of file
        .
        .
error := WRITEEDIT ( filenumber, record^num, line^image,
                     line^length );
```

Related Programming Manual

For programming information about the WRITEEDIT procedure, see the *Guardian Programmer's Guide*.

WRITEEDITP Procedure

Summary

The WRITEEDITP procedure accepts a line in EDIT packed line format and writes it to the specified file.

WRITEEDITP is an IOEdit procedure and can only be used with files that have been opened by OPENEDIT or OPENEDIT_.

Syntax for C Programmers

```
#include <cextdecs(WRITEEDITP)>

short WRITEEDITP ( short filenum
                  ,[ __int32_t record-number ]
                  ,const char *packed-line
                  ,short packed-length );
```

- CEXTDECS (through the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.

Syntax for TAL Programmers

```
error := WRITEEDITP ( filenum                    ! i
                     ,[ record-number ]          ! i
                     ,packed-line                ! i
                     ,packed-length );           ! i
```

Parameters

error returnedvalue

INT

returns a file-system error number indicating the outcome of the operation. Possible values include:

10 File already includes a line with the specified record number.
21 Invalid length specified in the *packed-length* parameter.
45 All of the file's possible extents are allocated and full. You can use EXTENDEDIT to increase the file's extent size and call WRITEEDITP again.

filenum input

INT:value

specifies the file number of the open file to which the line is to be written.

record-number input

INT(32):value

if present, specifies the record number of the line to be written. If *record-number*:

- is greater than or equal to 0, it specifies 1000 times the EDIT line number of the line to be written.
- is -1, the line is written at the beginning of the file.

- is -2, the line is written at the end of the file.
- is -3 or omitted, the line is written to the line represented by the file's current record number.

packed-line input

STRING .EXT:ref:*

is a string array that contains the line in packed format that is to be written. The length of *packed-line* is specified by the *packed-length* parameter.

packed-length input

INT:value

specifies the length in bytes of *packed-line*. The *packed-length* must be in the range 1 through 256.

Example

```
INT(32) record^num := -2D;  ! write to end of file
        .
        .
error := WRITEEDITP ( filenumber, record^num, line^image,
                      line^length );
```

Related Programming Manual

For programming information about the WRITEEDITP procedure, see the *Guardian Programmer's Guide*.

WRITEREAD[X] Procedures

Summary

The WRITEREAD[X] procedures write data to a file from an array in the application process, then waits for data to be transferred back from the file. The data from the read portion returns in the same array used for the write portion.

WRITEREAD is intended for use with 16-bit addresses, while WRITEREADX is intended for use with 32-bit extended addresses. Therefore, the data buffer for WRITEREADX can be either in the caller's stack segment or any extended data segment.

If the file is opened for nowait I/O, you must not modify the buffer before the I/O completes with a call to AWAITIOX. This condition also applies to other processes that may be sharing the segment. The application must ensure that the buffer used in the call to WRITEREADX is not reused before the I/O completes with a call to AWAITIOX.

Terminals A special hardware feature is incorporated in the asynchronous multiplexer controller, which ensures that the system is ready to read from the terminal as soon as the write is completed.

Interprocess Communication
The WRITEREAD[X] procedure is used to originate a message to another process which was previously opened, then waits for a reply from that process.

Syntax for C Programmers

```
#include <cextdecs(WRITEREAD)>

_cc_status WRITEREAD ( short filenum
                      ,short _near *buffer
                      ,unsigned short write-count
                      ,unsigned short read-count
                      ,[unsigned short _near *count-read ]
                      ,[ __int32_t tag ] );


#include <cextdecs(WRITEREADX)>

_cc_status WRITEREADX ( short filenum
                       ,char _far *buffer
                       ,unsigned short write-count
                       ,unsigned short read-count
                       ,[unsigned short _far *count-read ]
                       ,[ __int32_t tag ] );
```

- CEXTDECS (through the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.
- The function value returned by WRITEREAD[X], which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL WRITEREAD[X] ( filenum                          ! i
                  ,buffer                            ! i,o
                  ,write-count                       ! i
                  ,read-count                        ! i
                  ,[ count-read ]                    ! o
                  ,[ tag ] );                        ! i
```

Parameters

filenum input

INT:value (Use with WRITEREAD and WRITEREADX)

is the number of an open file that identifies the file where the WRITE/READ is to occur.

buffer input,output

INT:ref:* (Use with WRITEREAD)
STRING .EXT:ref:* (Use with WRITEREADX)

is an array containing information to be written to the file.

On return, *buffer* contains the information read from the file.

write-count input

INT:value (Use with WRITEREAD and WRITEREADX)

is the number of bytes to be written:

{0:32755} for terminals
{0:57344} for interprocess files

Note. When using terminals in block mode, an error 21 occurs if *write-count* exceeds 256 bytes.

---

read-count input

INT:value (Use with WRITEREAD and WRITEREADX)

returns the number of bytes to be read:

{0:32755} for terminals
{0:57344} for interprocess files

count-read output

INT:ref:1 (Use with WRITEREAD)
INT .EXT:ref:1 (Use with WRITEREADX)

is for wait I/O only. It returns a count of the number of bytes returned from the file into *buffer*.

tag input

INT(32):value (Use with WRITEREAD and WRITEREADX)

is for nowait I/O only. *tag* must uniquely identify the operation associated with this WRITEREAD[X].

Note. The system stores this *tag* value until the I/O operation completes. The system then returns the *tag* information to the program in the *tag* parameter of the call to AWAITIO[X], thus indicating that the operation completed. If WRITEREADX is used, you must call AWAITIOX to complete the I/O. If WRITEREAD is used, you can use either AWAITIO or AWAITIOX to complete the I/O.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILE_GETINFO_ or FILEINFO).

= (CCE) indicates the WRITEREAD[X] is successful.

> (CCG) indicates that CTRL-Y is pressed on the terminal.

Considerations

- WRITEREAD versus WRITEREADX

  Use WRITEREAD when the buffer has a 16-bit address, and use WRITEREADX when the buffer has a 32-bit extended address. Therefore, the data buffer for WRITEREADX can be either in the caller's stack segment or any extended data segment.

- Waited I/O READ

  If a waited I/O WRITEREAD[X] is executed, the *count-read* parameter indicates the number of bytes actually read.

- Nowait I/O READ

  If a nowait I/O WRITEREAD[X] is executed, *count-read* has no meaning and can be omitted. The count of the number of bytes read is obtained when the I/O operation completes through the *count-transferred* parameter of the AWAITIO[X] procedure.

  The WRITEREAD[X] procedure must complete with a corresponding call to the AWAITIO[X] procedure when used with a file that is opened nowait. If WRITEREADX is used, you must call AWAITIOX to complete the I/O. If WRITEREAD is used, you can use either AWAITIO or AWAITIOX to complete the I/O.

▲ **WARNING.** When using nowait file I/O, data corruption might occur if the READ or WRITE buffers are modified before the AWAITIOX that completes the call.

You should not change the contents of the data buffer between the initiation and completion of a nowait WRITEREAD operation. This is because a retry can copy

the data again from the user buffer and cause the wrong data to be written. You should avoid sharing a buffer between a WRITEREAD and another I/O operation because this creates the possibility of changing the contents of the data buffer before the write is completed.

- Carriage return/line feed sequence after the write

  There is no carriage return/line feed sequence sent to the terminal after the write part of the operation.

Considerations for WRITEREADX Only

- The buffer and count transferred may be in the user stack or in an extended data segment. The buffer and count transferred cannot be in the user code space.
- If the buffer or count transferred is in a selectable extended data segment, the segment must be in use at the time of the call. Flat segments allocated by a process are always accessible to the process.
- If the file is opened for nowait I/O, and the buffer is in an extended data segment, you cannot deallocate or reduce the size of the extended data segment before the I/O completes with a call to AWAITIOX or is canceled by a call to CANCEL or CANCELREQ.
- If the file is opened for nowait I/O, and the I/O has been initiated with these routines, the I/O must be completed with a call to AWAITIOX (not AWAITIO).
- If the file is opened for nowait I/O, the extended segment containing the buffer need not be in use at the time of the call to AWAITIOX.
- Nowait I/O initiated with these routines may be canceled with a call to CANCEL or CANCELREQ. The I/O is canceled if the file is closed before the I/O completes or AWAITIOX is called with a positive time limit and specific file number and the request times out.
- If the extended address of the buffer is odd, bounds checking rounds the address to the next lower word boundary and checks an extra byte as well. The odd address is used for the transfer.

Errors for WRITEREADX Only

In addition to the errors currently returned from the WRITEREAD procedure, file-system error 22 is returned when:

- The address of a parameter refers to the selectable segment area but no selectable segment is in use at the time of the call.
- The address of a parameter is extended, but it is an absolute address and the caller is not privileged.
- The file system cannot use the user's segment when needed.

Example

```
CALL WRITEREAD ( FILE^NUM, INOUT^BUFFER, 1, 72, NUM^READ );
```

The INOUT^BUFFER contains the information to be written, and after the write it contains information that was read. In this case, 1 byte is to be written, and 72 bytes are to be read. NUM^READ indicates how many bytes are read into the INOUT^BUFFER.

Related Programming Manuals

For programming information about the WRITEREAD file-system procedure, see the *Guardian Programmer's Guide*, the *Enscribe Programmer's Guide*, and the data communication manuals.

WRITEUPDATE[X] Procedures

Summary

The WRITEUPDATE[X] procedures transfer data from an array in the application program to a file.

WRITEUPDATE is intended for use with 16-bit addresses, while WRITEUPDATEX is intended for use with 32-bit extended addresses. Therefore, the data buffer for WRITEUPDATEX can be either in the caller's stack segment or any extended data segment.

For disk files, WRITEUPDATE[X] has two functions:

1. To alter the contents of the record at the current position
2. To delete the record at the current position in a key-sequenced or relative file

WRITEUPDATE[X] is used for processing data at random. Data from the application process's array is written in the position indicated by the setting of the current-record pointer. A call to this procedure typically follows a corresponding call to the READ[X]

or READUPDATE[X] procedure. The current-record and next-record pointers are not affected by the WRITEUPDATE[X] procedure.

For magnetic tapes, WRITEUPDATE[X] is used to replace a record in an already written tape. The tape is backspaced one record; the data from the application process's array is written in that area.

Syntax for C Programmers

```
#include <cextdecs(WRITEUPDATE)>

_cc_status WRITEUPDATE ( short filenum
                       ,short _near *buffer
                       ,unsigned short write-count
                       ,[unsigned short _near *count-written ]
                       ,[ __int32_t tag ] );


#include <cextdecs(WRITEUPDATEX)>

_cc_status WRITEUPDATEX ( short filenum
                        ,const char *buffer
                        ,unsigned short write-count
                        ,[unsigned short _far *count-written ]
                          ,[ __int32_t tag ] );
```

- CEXTDECS (through the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.
- The function value returned by WRITEUPDATE[X], which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL WRITEUPDATE[X] ( filenum                          ! i
                    ,buffer                            ! i
                    ,write-count                       ! i
                    ,[ count-written ]                 ! o
                    ,[ tag ] );                        ! i
```

Parameters

filenum input

INT:value (Use with WRITEUPDATE and WRITEUPDATEX)

is a number of an open file that identifies the file to be written.

buffer input

INT:ref:* (Use with WRITEUPDATE)
STRING .EXT:ref:* (Use with WRITEUPDATEX)

is an array containing the information to be written to the file.

write-count input

INT:value (Use with WRITEUPDATE and WRITEUPDATEX)

is the number of bytes to be written to the file:

{0:4096} for disk files (see Disk File Considerations on page 16-8)
{0:32767} for magnetic tapes

For key-sequenced and relative files: 0 means delete the record.
For entry-sequenced files: 0 means anything <> the record's length is invalid.

count-written output

INT:ref:1 (Use with WRITEUPDATE)
INT .EXT:ref:1 (Use with WRITEUPDATEX)

is for wait I/O only. It returns a count of the number of bytes written to the file.

tag input

INT(32):value (Use with WRITEUPDATE and WRITEUPDATEX)

is for nowait I/O only. *tag* is a value you define that uniquely identifies the operation associated with this WRITEUPDATE[X].

The system stores this *tag* value until the I/O operation completes. The system returns the *tag* information back to the program in the *tag* parameter of the call to AWAITIO[X], thus indicating that the operation completed. If WRITEUPDATEX is used, you must call AWAITIOX to complete the I/O. If WRITEUPDATE is used, you can use either AWAITIO or AWAITIOX to complete the I/O.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILE_GETINFO_ or FILEINFO).

< (CCL) is returned following successful insertion or update of a record in a file with one or more insertion-ordered alternate keys if a duplicate key value was created for at least one insertion-ordered alternate key. A call to FILE_GETINFO_ or FILEINFO shows that error 551 occurred; this error is advisory only and does not indicate an unsuccessful write operation.

= (CCE) indicates the WRITEUPDATE[X] was successful.

> (CCG) indicates that an error occurred (call FILE_GETINFO_ or FILEINFO).

Considerations

- WRITEUPDATE versus WRITEUPDATEX

Use WRITEUPDATE when the buffer has a 16-bit address, and use WRITEUPDATEX when the buffer has a 32-bit extended address. Therefore, the data buffer for WRITEUPDATEX can be either in the caller's stack segment or any extended data segment.

- I/O counts with unstructured files

  Unstructured files are transparently blocked using one of the four valid blocksizes (512, 1024, 2048, or 4096 bytes; 4096 is the default). This transparent blocksize, known as BUFFERSIZE, is the transfer size used against an unstructured file. While BUFFERSIZE does not change the maximum unstructured transfer (4096 bytes), multiple I/O operations might be performed to satisfy a user's request depending on the BUFFERSIZE chosen. For example, if BUFFERSIZE is 512 bytes, and a request is made to write 4096 bytes, at least eight transfers, each 512 bytes long, will be made. More than eight transfers happen, in this case, if the requested transfer does not start on a BUFFERSIZE boundary.

  DP2 performance with unstructured files is best when requested transfers begin on BUFFERSIZE boundaries and are integral multiples of BUFFERSIZE.

  Because the maximum blocksize for DP2 structured files is also 4096 bytes, this is also the maximum structured transfer size for DP2.

- Deleting locked records

  Deleting a locked record implicitly unlocks that record unless the file is audited, in which case the lock is not removed until the transaction terminates.

- Waited WRITEUPDATE[X]

  If a waited WRITEUPDATE[X] is executed, the *count-written* parameter indicates the number of bytes actually written.

- Nowait WRITEUPDATE[X]

  If a nowait WRITEUPDATE[X] is executed, *count-written* has no meaning and can be omitted. The count of the number of bytes written is obtained through the *count-transferred* parameter of the AWAITIO[X] procedure when the I/O completes.

  The WRITEUPDATE[X] procedure must finish with a corresponding call to the AWAITIO[X] procedure when used with a file that is opened nowait. For files audited by the Transaction Management Facility (TMF), the AWAITIO[X] procedure must be called before the ENDTRANSACTION or ABORTTRANSACTION procedure is called.

▲ **WARNING.** When using nowait file I/O, data corruption might occur if the READ or WRITE buffers are modified before the AWAITIOX that completes the call.

You should not change the contents of the data buffer between the initiation and completion of a nowait write operation. This is because a retry can copy the data again from the user buffer and cause the wrong data to be written. You should

avoid sharing a buffer between a write and another I/O operation because this creates the possibility of changing the contents of the write buffer before the write is completed.

- Invalid write operations to queue files

  Attempts to perform WRITEUPDATE[X] operations are rejected with an error 2.

Disk File Considerations

- Large data transfers

  For WRITEUPDATEX only, large data transfers (more than 4096 bytes), can be enabled by using SETMODE function 141. See Table 14-4 on page 14-63.

- Random processing and WRITEUPDATE[X]

  For key-sequenced, relative, and entry-sequenced files, random processing implies that a designated record must exist. This means that positioning for WRITEUPDATE[X] is always to the record described by the exact value of the current key and current-key specifier. If such a record does not exist, the call to WRITEUPDATE[X] is rejected with file-system error 11 (record does not exist).

- File is locked

  If a call to WRITEUPDATE[X] is made and the file is locked through a file number other than that supplied in the call, the call is rejected with file-system error 73 (file is locked).

- When the just-read record is updated

  A call to WRITEUPDATE[X] following a call to READ[X], without intermediate positioning, updates the record just read.

- Unstructured files

  - Unstructured disk file: transferring data

    If the WRITEUPDATE[X] is to an unstructured disk file, data is transferred to the record location specified by the current-record pointer.

  - File pointers after a successful WRITEUPDATE[X]

    After a successful WRITEUPDATE[X] to an unstructured file, the current-record and next-record pointers are unchanged.

  - The number of bytes written

    If the unstructured file is created with the odd unstructured attribute (also known as ODDUNSTR) set, the number of bytes written is exactly the number specified in *write-count*. If the odd unstructured attribute is not set when the file is created, the value of *write-count* is rounded up to an even value before the WRITEUPDATE[X] is executed.

You set the odd unstructured attribute with the FILE_CREATE_, FILE_CREATELIST_, or CREATE procedure, or with the File Utility Program (FUP) SET and CREATE commands.

- Structured files
  - Calling WRITEUPDATE[X] after KEYPOSITION

    If the call to WRITEUPDATE[X] immediately follows a call to KEYPOSITION in which a nonunique alternate key is specified as the access path, the WRITEUPDATE[X] fails. A subsequent call to FILE_GETINFO_ or FILEINFO shows that error 46 (invalid key) occurred. However, if an intermediate call to READ[X] or READLOCK[X] is performed, the call to WRITEUPDATE[X] is permitted because a unique record is identified.

  - Specifying *write-count* for entry-sequenced files

    For entry-sequenced files, the value of *write-count* must match exactly the *write-count* value specified when the record was originally inserted into the file.

  - Changing the *primary-key* of a key-sequenced record

    An update to a record in a key-sequenced file cannot alter the value of the *primary-key* field. Changing the *primary-key* field must be done by deleting the old record (WRITEUPDATE[X] with *write-count* = 0) and inserting a new record with the key field changed (WRITE).

  - Current-state indicators after WRITEUPDATE[X]

    After a successful WRITEUPDATE[X], the current-state indicators remain unchanged.

Magnetic Tape Considerations

- WRITEUPDATE[X] is permitted only on the 3202 Controller for the 5103 or 5104 Tape Drives. This command is not supported on any other controller/tape drive combination.

  WRITEUPDATE[X] is specifically not permitted on these controller/tape drive pairs:

  3206 Controller and the 5106 Tri-Density Tape Drive
  3207 Controller and the 5103 & 5104 Tape Drives
  3208 Controller and the 5130 & 5131 Tape Drives

- Specifying the correct number of bytes written

  When WRITEUPDATE[X] is used with magnetic tape the number of bytes to be written must fit exactly; otherwise, information on the tape can be lost. However, no error indication is given.

- Limitation of WRITEUPDATE[X] to the same record

  Five is the maximum number of times a WRITEUPDATE[X] can be executed to the same record on tape.

Considerations for WRITEUPDATEX Only

- The buffer and count transferred may be in the user stack or in an extended data segment. The buffer and count transferred cannot be in the user code space.
- If the buffer or count transferred is in a selectable extended data segment, the segment must be in use at the time of the call. Flat segments allocated by a process are always accessible to the process.
- If the file is opened for nowait I/O, and the buffer is in an extended data segment, you cannot deallocate or reduce the size of the extended data segment before the I/O completes with a call to AWAITIOX or is canceled by a call to CANCEL or CANCELREQ.
- If the file is opened for nowait I/O, you must not modify the buffer before the I/O completes with a call to AWAITIOX. This also applies to other processes that may be sharing the segment. It is the application's responsibility to ensure this.
- If the file is opened for nowait I/O, and the I/O has been initiated with these routines, the I/O must be completed with a call to AWAITIOX (not AWAITIO).
- If the file is opened for nowait I/O, the extended segment containing the buffer need not be in use at the time of the call to AWAITIOX.
- Nowait I/O initiated with these routines may be canceled with a call to CANCEL or CANCELREQ. The I/O is canceled if the file is closed before the I/O completes or AWAITIOX is called with a positive time limit and specific file number and the request times out.
- If the extended address of the buffer is odd, bounds checking rounds the address to the next lower word boundary and checks an extra byte as well. The odd address is used for the transfer.

Errors for WRITEUPDATEX Only

In addition to the errors returned from the WRITEUPDATE procedure, file-system error 22 is returned when:

- The segment is in use at the time of the call or the segment in use is invalid.
- The address of a parameter is extended, but it is an absolute address and the caller is not privileged.
- The file system cannot use the user's segment when needed.

Example

```
CALL WRITEUPDATE ( TAPE^NUM , TAPE^BUF , NUM^READ , NUM^WRITTEN
);
```

The application makes the necessary changes to the record in TAPE^BUF, then edits the tape by calling WRITEUPDATE. The tape is backspaced over the record just read, then updated by writing the new record in its place. NUM^READ indicates the number of bytes to be written (ensuring that the same number of bytes just read are also written).

Related Programming Manuals

For programming information about the WRITEUPDATE file-system procedure, see the *Enscribe Programmer's Guide* and the *Guardian Programmer's Guide*.

WRITEUPDATEUNLOCK[X] Procedures

Summary

The WRITEUPDATEUNLOCK[X] procedures perform random processing of records in a disk file.

WRITEUPDATEUNLOCK is intended for use with 16-bit addresses, while WRITEUPDATEUNLOCKX is intended for use with 32-bit addresses. Therefore, the data buffer for WRITEUPDATELOCKX can be either in the caller's stack segment or any extended data segment.

WRITEUPDATEUNLOCK[X] has two functions:

1. To alter, then unlock, the contents of the record at the current position
2. To delete the record at the current position in a key-sequenced or relative file

A call to WRITEUPDATEUNLOCK[X] is equivalent to a call to WRITEUPDATE[X] followed by a call to UNLOCKREC. However, the WRITEUPDATEUNLOCK[X] procedure requires less system processing than do the separate calls to WRITEUPDATE[X] and UNLOCKREC.

Syntax for C Programmers

```
#include <cextdecs(WRITEUPDATEUNLOCK)>

_cc_status WRITEUPDATEUNLOCK ( short filenum
                      ,short _near *buffer
                      ,unsigned short write-count
                      ,[unsigned short _near *count-written ]
                      ,[ __int32_t tag ] );


#include <cextdecs(WRITEUPDATEUNLOCKX)>

_cc_status WRITEUPDATEUNLOCKX ( short filenum
                       ,const char _far *buffer
                       ,unsigned short write-count
                       ,[unsigned short _far *count-written ]
                         ,[ __int32_t tag ] );
```

- CEXTDECS (through the included file TNSINTH) defines 32-bit values as the *typedef* __int32_t which for TNS and TNS/R compiles is defined as *long* and for TNS/E compiles is defined as *int*.
- The function value returned by WRITEUPDATEUNLOCK[X], which indicates the condition code, can be interpreted by `_status_lt()`, `_status_eq()`, or `_status_gt()` (defined in the file `tal.h`).

Syntax for TAL Programmers

```
CALL WRITEUPDATEUNLOCK[X] ( filenum                    ! i
                           ,buffer                     ! i
                           ,write-count                ! i
                           ,[ count-written ]          ! o
                           ,[ tag ] );                 ! i
```

Parameters

filenum input

INT:value (Use with WRITEUPDATEUNLOCK and WRITEUPDATEUNLOCKX)

is a number of an open file that identifies the file to be written.

buffer input

INT:ref:* (Use with WRITEUPDATEUNLOCK)
STRING .EXT:ref:* (Use with WRITEUPDATEUNLOCKX)

is an array containing the data to be written to the file.

write-count input

INT:value (Use with WRITEUPDATEUNLOCK and WRITEUPDATEUNLOCKX)

is the number of bytes to be written to the file: {0:4096}.

For key-sequenced and relative files 0 deletes the record
For entry-sequenced files 0 is invalid (error 21)

count-written output

INT:ref:1 (Use with WRITEUPDATEUNLOCK)
INT .EXT:ref:1 (Use with WRITEUPDATEUNLOCKX)

is for wait I/O only. It returns an integer indicating the number of bytes written to the file.

tag input

INT(32):value (Use with WRITEUPDATEUNLOCK and WRITEUPDATEUNLOCKX)

is for nowait I/O only. *tag* is a value you define that uniquely identifies the operation associated with this WRITEUPDATEUNLOCK[X].

Note. The system stores this *tag* value until the I/O operation completes. The system then returns the *tag* information to the program in the *tag* parameter of the call to AWAITIO[X], thus indicating that the operation completed. If WRITEUPDATEUNLOCKX is used, you must call AWAITIOX to complete the I/O. If WRITEUPDATEUNLOCK is used, you can use either AWAITIO or AWAITIOX to complete the I/O.

Condition Code Settings

< (CCL) indicates that an error occurred (call FILE_GETINFO_ or FILEINFO).

< (CCL) is returned following successful insertion or update of a record in a file with one or more insertion-ordered alternate keys if a duplicate key value was created for at least one insertion-ordered alternate key. A call to FILE_GETINFO_ or FILEINFO shows that error 551 occurred; this error is advisory only and does not indicate an unsuccessful write operation.

= (CCE) indicates that the WRITEUPDATEUNLOCK[X] was successful.

> (CCG) indicates that an error occurred (call FILE_GETINFO_ or FILEINFO).

Considerations

- WRITEUPDATEUNLOCK versus WRITEUPDATEUNLOCKX

  Use WRITEUPDATEUNLOCK when the buffer has a 16-bit address, and use WRITEUPDATEUNLOCKX when the buffer has a 32-bit extended address. Therefore, the data buffer for WRITEUPDATEUNLOCKX can be either in the caller's stack segment or any extended data segment.

- Nowait I/O and WRITEUPDATEUNLOCK[X]

  The WRITEUPDATEUNLOCK[X] procedure must complete with a corresponding call to the AWAITIO[X] procedure when used with a file that is opened nowait. If WRITEUPDATEUNLOCKX is used, you must call AWAITIOX to complete the I/O. If WRITEUPDATEUNLOCK is used, you can use either AWAITIO or AWAITIOX to complete the I/O.

  For files audited by the Transaction Management Facility (TMF), AWAITIO[X] must be called to complete the WRITEUPDATEUNLOCK[X] operation before ENDTRANSACTION or ABORTTRANSACTION is called.

▲ **WARNING.** When using nowait file I/O, data corruption might occur if the READ or WRITE buffers are modified before the AWAITIOX that completes the call.

  You should not change the contents of the data buffer between the initiation and completion of a nowait write operation. This is because a retry can copy the data again from the user buffer and cause the wrong data to be written. You should avoid sharing a buffer between a write and another I/O operation because this creates the possibility of changing the contents of the write buffer before the write is completed.

- Random processing and WRITEUPDATEUNLOCK[X]

  For key-sequenced, relative, and entry-sequenced files, random processing implies that a designated record must exist. This means positioning for WRITEUPDATEUNLOCK[X] is always to the record described by the exact value of the current key and current-key specifier. If such a record does not exist, the call to WRITEUPDATEUNLOCK[X] is rejected with file-system error 11 (record does not exist).

- Unstructured files—pointers unchanged

  For unstructured files, data is written in the position indicated by the current-record pointer. A call to WRITEUPDATEUNLOCK[X] for an unstructured file typically follows a call to POSITION or READUPDATE[X]. The current-record and next-record pointers are not changed by a call to WRITEUPDATEUNLOCK[X].

- How WRITEUPDATEUNLOCK[X] works

  The record unlocking performed by WRITEUPDATEUNLOCK[X] functions in the same manner as UNLOCKREC.

- Record does not exist

Positioning for WRITEUPDATEUNLOCK[X] is always to the record described by the exact value of the current key and current-key specifier. Therefore, if such a record does not exist, the call to WRITEUPDATEUNLOCK[X] is rejected with file-system error 11.

See Considerations on page 16-26.

- Invalid write operations to queue files

  DP2 rejects WRITEUPDATEUNLOCK[X] operations with an error 2.

Considerations for WRITEUPDATEUNLOCKX Only

- The buffer and count transferred may be in the user stack or in an extended data segment. The buffer and count transferred cannot be in the user code space.
- If the buffer or count transferred is in a selectable extended data segment, the segment must be in use at the time of the call. Flat segments allocated by a process are always accessible to the process.
- If the file is opened for nowait I/O, and the buffer is in an extended data segment, you cannot deallocate or reduce the size of the extended data segment before the I/O completes with a call to AWAITIOX or is canceled by a call to CANCEL or CANCELREQ.
- If the file is opened for nowait I/O, you must not modify the buffer before the I/O completes with a call to AWAITIOX. This also applies to other processes that may be sharing the segment. It is the application's responsibility to ensure this.
- If the file is opened for nowait I/O, and the I/O has been initiated with these routines, the I/O must be completed with a call to AWAITIOX (not AWAITIO).
- If the file is opened for nowait I/O, the extended segment containing the buffer need not be in use at the time of the call to AWAITIOX.
- Nowait I/O initiated with these routines may be canceled with a call to CANCEL or CANCELREQ. The I/O is canceled if the file is closed before the I/O completes or AWAITIOX is called with a positive time limit and specific file number and the request times out.
- If the extended address of the buffer is odd, bounds checking rounds the address to the next lower word boundary and checks an extra byte as well. The odd address is used for the transfer.

Errors for WRITEUPDATEUNLOCKX Only

In addition to the errors returned from the WRITEUPDATEUNLOCK procedure, file-system error 22 is returned when:

- The address of a parameter is extended, but no segment is in use at the time of the call or the segment in use is invalid.

- The address of a parameter is extended, but it is an absolute address and the caller is not privileged.
- The file system cannot use the user's segment when needed.

OSS Considerations

- This procedure operates only on Guardian objects. If an OSS file is specified, error 2 is returned.

Example

```
CALL WRITEUPDATEUNLOCK ( OUT^FILE , OUT^BUFFER , 72   &
                         , NUM^WRITTEN );
```

Related Programming Manuals

For programming information about the WRITEUPDATEUNLOCK file-system procedure, see the *Enscribe Programmer's Guide* and the *Guardian Programmer's Guide*.

XBNDSTEST Procedure (Superseded by REFPARAM_BOUNDSCHECK_ Procedure)

Summary

Note. This procedure cannot be called by native processes. Although this procedure is supported for TNS processes, it should not be used for new development.

XBNDSTEST aids user programs in checking stack limits or parameter addresses. (LASTADDRX provides a similar function.) XBNDSTEST uses constants obtained from another procedure, XSTACKTEST, to check a specified address and its length for potential bounds violations.

Syntax for C Programmers

```
#include <cextdecs(XBNDSTEST)>

short XBNDSTEST ( char *param
                 ,short bytelen
                 ,short flags
                 ,long long constants );
```

Syntax for TAL Programmers

```
status := XBNDSTEST ( param                ! i
                     ,bytelen              ! i
                     ,flags                ! i
                     ,constants );         ! i
```

Parameters

status returnedvalue

INT

returns one of these values:

1 In bounds, but in a read-only segment or (on native systems only) in the system library
0 In bounds and writable

-1 Out of bounds or invalid address
-2 Incorrectly aligned on word boundary
-3 Undefined flag value
-4 In bounds, but in an extensible extended data segment that cannot be extended (usually due to lack of additional disk space for the swap file)

param input

STRING .EXT:ref:*

is the parameter to be bounds-checked.

bytelen input

INT:value

is the unsigned length of the parameter, in bytes {0:65535}.

flags input

INT:value

is defined as:

<0:12>	must be zero
<13>	use extended address limits from *constants*
<14>	parameter must be word-aligned
<15>	skip the bounds test and return 0

constants input

FIXED:value

is a set of constant values generated by XSTACKTEST.

Considerations

- XBNDSTEST can perform extended address checking against either the current extended address limit or the limit in effect at the time XSTACKTEST was called. The latter may be specified by setting bit <13> of the *flags* parameter.
- XBNDSTEST will normally reject all relative extended address references to the system data segment (segment 1) as well as all absolute extended addresses. Procedures that support privileged callers may disable these checks by either:
  - setting bit <15> of the *flags* parameter, which will disable all address checking.
  - calling XSTACKTEST with bit <14> of the *flags* parameter set, which will generate a *constants* value that permits privileged mode addressing but still performs the normal checks on other addresses.
- Before the D20 RVU, these *status* conditions were handled differently:

- An address falling in the system library area on a native system, which now causes a *status* value of 1 to be returned, previously caused a *status* value of -1 (invalid address) to be returned.
- The condition that now causes a *status* value of -4 to be returned (address is in an extensible extended data segment that cannot be extended) previously caused a *status* value of -1 (invalid address) to be returned.

XSTACKTEST Procedure (Superseded by HEADROOM_ENSURE_ Procedure)

Summary

Note. This procedure cannot be called by native processes. Although this procedure is supported for TNS processes, it should not be used for new development.

XSTACKTEST, used with LASTADDRX and XBNDSTEST checks stack limits. XSTACKTEST ensures that adequate stack space is available and returns a set of constants to be used with the XBNDSTEST procedure.

Syntax for C Programmers

```
#include <cextdecs(XSTACKTEST)>

short XSTACKTEST ( short _near *firstparam
                  ,short stackwords
                  ,short flags
                  ,long long *constants );
```

Syntax for TAL Programmers

```
status := XSTACKTEST ( firstparm              ! i
                      ,stackwords             ! i
                      ,flags                  ! i
                      ,constants );           ! o
```

Parameters

status returnedvalue

INT

returns one of these file-system error numbers:

0	Adequate stack space available
2	Undefined *flags* value
21	*stackwords* is less than 1
22	Bounds error on *firstparm* or *constants*
632	Insufficient stack space available

firstparm input

INT:ref:*

points to the first parameter word of the first called procedure.

stackwords input

INT:value

is the number of words required for the stack, starting from the *firstparm* location.

flags input

INT:value

is defined as:

<0:13>	must be zero
<14>	allow parameters that use privileged addresses
<15>	skip the stack test and return 0

constants output

FIXED:ref:1

is a returned set of constant values that may be used when calling XBNDSTEST.

Considerations

- If bit <14> of the *flags* parameter is set, the *constants* value generated by XSTACKTEST causes XBNDSTEST to accept extended addresses that refer to the system data segment or that use absolute extended addressing.
- If bit <15> of the *flags* parameter is set, XSTACKTEST immediately returns 0 and the *constants* parameter is NOT modified.

A Device Types and Subtypes

Table A-1. Device Types and Subtypes (page 1 of 9)

Type	Device	Sub type	G-Series and H-Series Descriptions
0	Process	0	Default subtype for general use
		1-49	Reserved for definition by HP. These subtypes are defined: 1 = CMI process 2 = Security monitor process 30 = Device simulation process 31 = Spooler collector process 48 = TFTP server process 49 = SNMP trap multiplexor
		50-63	For general use
1	Operator console	0	$0 (operator process) or alternate collector
		1	$0.#ZSPI ($0 opened to receive SPI commands)
		2	$Z0 (compatibility distributor)
2	$RECEIVE	0	
3	Disk	2	N.A.
		3	N.A.
		4	N.A.
		5	N.A.
		6	N.A.
		7	N.A.
		8	N.A.
			N.A.
		9	N.A.
		10	N.A.
		16	N.A.
		17	N.A.
		18	N.A.
		19	N.A.
		20	N.A.
		21	N.A.
		22	N.A.
		23	N.A.
		26	N.A.

Table A-1. Device Types and Subtypes (page 2 of 9)

Type	Device	Sub type	G-Series and H-Series Descriptions
		29	N.A.
		31	N.A.
		33	N.A.
		34	N.A.
		36	NonStop™ Storage Management Foundation (SMF) virtual disk process
		38	4560 (2 GB formatted capacity per spindle) with ServerNet/DA
		39	4570 (4 GB) with ServerNet DA
		41	4604 (4 GB)
		42	4608 (8 GB)
			4609 (8 GB)
		43	4618 (18 GB)
		43	4619 (18 GB) (15,000 rpm)
		44	4636 (36 GB) 4637 (36 GB)
		45	4672 (72 GB)
		46	46144 (144 GB) SCSI disk for an S-series system
		48	4590 (18 GB) with ServerNet/DA
		52	ESS attached disk of dynamic variable size
		53	JBOD attached disk of variable size
		56	N.A.
4	Magnetic tape unit	0	N.A.
		1	N.A.
		2	N.A.
		3	N.A.
		4	N.A.
		5	N.A.
		6	5170 tape unit (1600, 6250 bpi) PMF/IOMF or ServerNet/DA
		7	N.A.
		8	N.A.
		9	5190 tape unit (18 track, 38000 bpi) or 5194 tape unit (36 tracks, 38000 bpi) with PMF, IOMF, or ServerNet/DA

Table A-1. Device Types and Subtypes (page 3 of 9)

Type	Device	Sub type	G-Series and H-Series Descriptions
		10	N.A.
		11	5142 DAT with PMF or IOMF
4	Magnetic	14	521A, 524A, 525A (LTO) tape units (512 tracks, 7.32kb/mm) with PMF or IO
5	Printer	0	
		1	N.A.
		3	
		4	
		5	N.A.
		6	
		7	
		8	
		9	
		10	
		32	
6	Terminal	0	Conversational mode (P/N 6401/6402) or PATPTERM (non-HP) device
		1	Page mode (P/N 6511, 6512)
		2	Page mode (P/N 6520, 6524)
		3	N.A.
		4	Page mode (P/N 6526, 6528, 653x)
		5	N.A.
		6-10	Conversational mode
			6 = 3277 (screen size 12x40)
			7 = 3277 (screen size 24x80)
			8 = 3277 (screen size 32x80)
			9 = 3277 (screen size 43x80)
			10 = 3277 (screen size 12x80)
		11	6340 FaxLink
		16	
	SNAX Interactive Terminal Interface (ITI) Protocol	20	3275-11, 3276-1 & -11, 3277-1, 3278-1

Table A-1. Device Types and Subtypes (page 4 of 9)

Type	Device	Sub type	G-Series and H-Series Descriptions
		21	3275-12, 3276-2 & -12, 3277-2, 3278-2, 3178-C10, -C20, -C3, & -C4, 3191-A1K & -A2K, 3279-2A, -2B, -S2A, -S2B, & -02X, 5578-001, -002, F-6652-A, & -C
		22	3276-3, 3278-3, 3277-3, 3279-3A, -3B, -S3G, & -03X
		23	3276-4 & -14, 3278-4, 3277-4, 6580-A04, -A06, -A08, & -A10
		24	3278-5
		30	3262, 3284, 3286, 3282, 3289
		32	6603/6604 terminal
7	Envoy data communications line	0	BISYNC, point-to-point, nonswitched
		1	BISYNC, point-to-point, switched
		2	BISYNC, multipoint, tributary
		3	BISYNC, multipoint, supervisor
		8	ADM-2, multipoint, supervisor
		9	TINET, multipoint, supervisor
		10	Burroughs, multipoint, supervisor
		11	Burroughs, point-to-point, contention
		13	Burroughs, point-to-point, contention
		30	Full duplex (FDX), out line
		31	Full duplex (FDX), in line
		32	NASDAQ, Full duplex (FDX), out line
		33	NASDAQ, Full duplex (FDX), in line
		40	Asynchronous line supervisor
		50	N.A.
		56	N.A.
8	Open SCSI		
9	Process-to-process interface	0	X25AM process

Table A-1. Device Types and Subtypes (page 5 of 9)

Type	Device	Sub type	G-Series and H-Series Descriptions
10	Terminal SNAX Cathode-Ray Tube (CRT) protocol	0	327x CRT mode Interface
		20	3275-11, 3276-1 & -11, 3277-1, 3278-1
		21	3275-12, 3276-2 & -12, 3277-2, 3278-2, 3178-C10, -C20, -C3, & -C4, 3279-2A, -2B, -S2A, -S2B, & -02X
		22	3276-3, 3278-3, 3277-3, 3279-3A, -3B, -S3G, & -03X
		23	3276-4 & -14, 3278-4, 3277-4, 6580-A04, -A06, -A08, & -A10
		24	3278-5, 3276-5, 3277-5
		30	3262, 3284, 3286, 3287, 3289
11	EnvoyACP/XF	40	FRMEXF or SDLCXF (synchronous data-link control) line
		41	HDLCXF (high-level data-link control) line
		42	ADCCP (advanced data communications control procedures) line
		43	Frame protocol
12	Tandem-to-IBM Link (TIL)	0	
13	SNAX/XF or SNAX/APN	5	SNASVM (Service Manager Process)
14	SNALU	0	SNA Application Logical Unit (SNALU)
15	SNAX/3501	0	3501 Data Encryption Devices
		1	Key manager (ZKEY)
		3	High performance security modules
	NSP	4	Atalla A6000 Network Security Processor
19	IPX/SPX	0	Manager process
		1	Protocol process
20-23	NTM/MP	0	NonStop Transaction Management Facility (TMF)
24	OSS		Open System Services
25	SMF pool	0	Storage pool process

Table A-1. Device Types and Subtypes (page 6 of 9)

Type	Device	Sub type	G-Series and H-Series Descriptions
26	Tandem HyperLink (THL)	0	
27	IPBMON	0	Interprocessor bus monitor for Fiber Optic Extension (FOX) or TorusNet vertical subsystem in FOX-compatibility mode
		5	$IPB1 (TorusNet vertical subsystem master service manager in multiple-link mode)
		6	Service manager for additional TorusNet vertical links
28	$ZNUP	0	Network Utility Process
29	$ZMIOP	1	Subsystem manager
30	Optical disk unit	0	N.A.
		1	N.A.
		2	N.A.
		3	N.A.
31	SNMP	0	NonStop SNMP Agent
36	TandemTalk	1	N.A.
		2	N.A.
		3	N.A.
		4	N.A.
37	ISDN	0	Integrated Services Digital Network Subsystem Manager
43	SLSA	0	ServerNet LAN Systems Access (SLSA) manager process
		1	ServerNet LAN Systems Access (SLSA) monitor process (LAN MON)
44	any device type > 63	0	44 is displayed as the type for any device with a device type greater than 63. The program is unable to return information on device types greater than 63. Use the newer Guardian procedures (those that are not superseded) to obtain information on device types that are greater than 63.
45	QIO	0	Queue I/O Monitor Process
46	TELNET	0	TELNET Server Process
48	TCP/IP	0	
49	SNAX/CDF	0	N.A.

Table A-1. Device Types and Subtypes (page 7 of 9)

Type	Device	Sub type	G-Series and H-Series Descriptions
50	CSM	0	N.A.
		1	N.A.
		2	SWAN concentrator manager (CONMGR)
		3	$ZZWAN WAN manager process
		4	WANBOOT process
		63	Subsystem Control Point (SCP)
51	CP6100	0	Line interface unit (LIU)
		1	Bisynchronous (BISYNC) point-to-point line
		2	ADCCP line
		3	N.A.
		4	MPSB Burroughs multipoint
52	SMF master	0	SMF master process
53	ATP6100	0	
		1	
		2	
54	DDNAM	0	
		63	
55	Open Systems Interconnection (OSI)	1	OSI/Application Services (OSI/AS) Manager
		4	Transport service provider (TSP)
		5	HP application, presentation, and session (TAPS) processes
		11	OSI/Message Handling System (OSI/MHS)
		12	OSI/Message Handling System (OSI/MHS)
		20	OSI/FTAM Application Manager
		21	OSI/FTAM Services
		24	OSI/CMIP
		25	OSI/FTAM Services
56	Multilan	0	N.A.
		1	N.A.
		2	N.A.
		3	N.A.
		4	N.A.

Table A-1. Device Types and Subtypes (page 8 of 9)

Type	Device	Sub type	G-Series and H-Series Descriptions
		5	N.A.
		6	N.A.
57	GDS	0	General Device Support
58	SNAX/XF or SNAX/APN	0	
		1	
		2	N.A.
		3	
		4	
59	AM6520	0	N.A.
		10	N.A.
60	AM3270	0	
		10	Line attached to SWAN concentrator
	TR3271	1	
		11	Line attached to SWAN concentrator
61	X.25	0-61	N.A.
		62	N.A.
		63	Line attached to SWAN concentrator
62	Expand NCP	6	$NCP Network Control Process
63	Expand Line Handler	0	Single-line handler over X.25, SNA, IP, ATM
		1	Multiline path handler
		2	Multiline line handler over X.25, SNA, IP, ATM
		3	Single-line handler over FOX (ServerNet/FX)
		4	Single-Line handler over ServerNet/Cluster
		5	Single-line handler SWAN_DIRECT or SWAN_SATELLITE
		6	Multiline line handler SWAN_DIRECT or SWAN_SATELLITE
63	Expand Manager	30	Expand Manager process $ZEXP
64	ServerNet Monitor	0	ServerNet/Cluster Monitor process $ZZSCL
		1	ServerNet/Cluster SANMAN process $ZZSMN
		2	ServerNet/Cluster MSGMON processes

Table A-1. Device Types and Subtypes (page 9 of 9)

Type	Device	Sub type	G-Series and H-Series Descriptions
65	Storage Subsystem Manager	0	Configures and controls storage I/O processes
66	NonStop Kernel Management	0	Configures and controls system-wide attributes and generic processes
67	SCSI Lock Management	0	Coordinates resource sharing SCSI I/O subsystem components across processors

B Reserved Process Names

This appendix contains the names that should be avoided when choosing process names. The names listed here are reserved for HP use:

$AOPR
$CMON
$CMP
$C9341
$DM*nn*
$IMON
$IPB
$KEYS
$MLOK
$NCP
$NULL
$OSP
$PM
$S
$SIM*nn*
$SPLS
$SSCP
$SYSTEM
$T
$TICS
$TMP
$X*name*
$Y*name*
$Z*name*

nn is any two digits (00 through 99).
name is any combination of 1 through 4 letters or digits (A through Z, 0 through 9).

These names are not reserved, but should be used with caution because they are commonly used for a specific purpose:

$DISC	$LP	$TAPE
$DISK	$SPLP	

C Completion Codes

This appendix lists the completion codes returned after execution of a process or, in some instances, a job step. These codes indicate the degree of success of a program in a standard manner, thus making it possible to create or build further steps based on these codes.

Completion codes -32768 through -1 are reserved for HP use. The caller must be privileged to have a negative completion code returned to its ancestor. Completion codes 0 through 999 are reserved and are shared by the customer and HP.

Completion codes from 1000 through +32767 are reserved for customers. HP subsystems will not use these completion codes.

These completion codes are defined and should be used according to these definitions for uniformity:

Completion Code	Definition
0	Normal, voluntary termination with no errors. This code is the default code for PROCESS_STOP_ (if abnormal termination is not specified) and STOP if no completion code is specified, and for the OSS exit() function if the exit status is 0.
1	Normal, voluntary termination with WARNING diagnostics. For example, if the process is a compiler, the compilation terminated with WARNING diagnostics after building a complete object file.
2	Abnormal, voluntary termination with FATAL errors or diagnostics. For example, if the process is a compiler, the compilation terminated with FATAL diagnostics and either an object file was not built or, if built, might be incomplete. A complete listing is generated.
3	Abnormal, voluntary, but premature termination with FATAL errors or diagnostics. For example, if the process is a compiler, the compilation terminated with FATAL diagnostics, with either no object file or an incomplete object file being built and an incomplete listing generated (the compiler quit compiling prematurely).
4	Process never got started. This completion code exists primarily for the use of the command interpreter or other command language interpreters that can act as the executor process of a batch job. This code allows the executor process to detect that a process associated with a RUN statement never got started. In that sense, this completion code is a “fake” completion code. The command interpreter acts as though it received a termination message from the process that it tried to create, when in fact it received an error returned by the procedure or OSS function that launched the process. The command interpreter then makes the completion code and the error returned by the procedure or OSS function that launched the process available for evaluation, for example, by a batch job executor process.

Completion Code	Definition
5	Process calls PROCESS_STOP_ (with abnormal termination specified) or ABEND on itself. This code is the default completion code for the PROCESS_STOP_ procedure (when abnormal termination is specified) and the ABEND procedure.
6	PROCESS_STOP_, STOP, or ABEND was called to delete a process by an external, but authorized, process. The system includes this completion code in the process deletion message. If the process cannot be stopped, the request is saved so that when the process calls SETSTOP this completion code is sent with the process deletion message. The user ID, the PCBCRAID (CAID) and the process ID of the process that caused the termination, are included in the termination message.
7	Restart this job. This completion code is used by the NetBatch scheduler and an executor process. The executor process sets its completion code to this value upon termination; the scheduler interprets this completion code and restarts a “restartable” job.
8	Code 8 is the same as code 1, normal termination, except that the user must examine the listing file to determine whether the results are acceptable. Completion code 8 is typically used by compilers.
9	The `kill()` or `raise()` function generated a signal that stopped the process. The termination information provides the signal number.

Note. If a signal is delivered to a signal handler that stops the process, the completion code will be determined by the handler. For example, when a signal stops a native C program, a different completion code is returned as set by the signal handler installed by the Common Run-Time Environment (CRE).

These completion codes are reserved for HP use:

Completion Code	Definition
-1	A trap was detected in a Guardian TNS process. If the system detects the absence of a trap handler routine or encounters another trap in a trap handler, then in addition to an abnormal termination, this completion code is returned automatically in the process deletion (ABEND) message. The contents of the text string vary with the state of the process. The first nine characters are "TRAPNO=*nn*" with *nn* representing the trap number in decimal. Then the text identifies the code space, including the TNS code segment index when appropriate, and indicates whether the process was privileged. Finally, the text displays key registers, depending upon the execution mode of the process at the time of its termination: P or pc, L, and S for TNS or accelerated mode; pc and sp for native mode. Examples: Invalid address in TNS mode: TRAPNO=00: (UC.00) P=%000012 L=%000001 S=%000003 Arithmetic overflow (division by zero) in accelerated mode, privileged: TRAPNO=02: (acc UC, Priv) pc=%h7042370C L=%023520 S=%023526 Limits-exceeded in native mode, privileged: TRAPNO=05: (SCr, Priv) pc=0x808E2EDC sp=0x5FFFFF00
-2	This code is returned by the system when a process has terminated itself but the system is unable to pass along the requested completion code and the associated termination information due to a resource problem in the system.
-3	This code is returned by the system when a process terminating itself passed bad parameters to PROCESS_DELETE_, STOP, or ABEND. In this case, some or all of the information requested in the completion code message may not be present. Because the process is stopping itself, it is stopped.
-4	This code is returned by the system when a processor failure caused the name of a process to be deleted (that is, the only process running under that name was in the processor that failed).
-5	A communications or resource failure occurred during the execution of one of the functions in the OSS `exec` or `tdm_exec` set of functions; or an initialization failure of the new process occurred when it was too late for the exec or `tdm_exec` function to return an error to its caller.

Completion Code	Definition
-6	An OSS process or native process terminated when it caused a hardware exception. The termination information field of the message contains the signal number. The termination text is in the message for all processes. However, while the TACL command interpreter displays the termination text when it is present in the message for a process created by TACL, OSS utilities such as `osh` typically do not. The text shows the signal number and name, identifies the code space, and indicates whether the process was privileged. For a native process, the text displays the pc and sp registers. For an OSS process, it shows registers appropriate to the mode, as for completion code -1. Examples: Invalid address in native mode: Arithmetic overflow (division by zero) in native mode, privileged: Signal 8, SIGFPE: (UCr, Priv) pc=0x70002D48 sp=0x5FFFFEB8
-7	An OSS process or native process terminated as a result of a corrupted stack frame or register state.
-8	An OSS process or native process terminated because of insufficient user stack space for signal delivery. Stack overflow generates completion code -8, which is otherwise like completion code -6. Example: Stack overflow in native mode: Signal 25, SIGSTK: (UCr) pc=0x70000394 sp=0x4FEFFE18
-9	An OSS process or native process terminated because of insufficient PRIV stack space for signal delivery. The termination information field of the message contains the signal number.
-10	An OSS process or native process terminated because it was unable to obtain resources for signal delivery. The termination information field of the message contains the signal number.

Completion Code	Definition
-11	An OSS process or native process terminated because it attempted to resume from a nonresumable signal. The termination information field of the message contains the signal number.
-12	One of the functions in the OSS `exec` or `tdm_exec` set of functions executed successfully. The OSS process ID continues to exist as it migrates to another process handle, but the original process handle is deleted. Call PROCESS_GETINFOLIST_ to obtain the new process handle of the OSS process.
-13	The OSS `open()` or `dup()` function performed by the PROCESS_SPAWN_ procedure failed. The termination information in *sysmsg*[17] contains the OSS `errno` for the error that occurred. The subsystem ID in *sysmsg*[18] contains the null value. The termination text in *sysmsg*[41] can contain additional information.

D

File Names and Process Identifiers

This appendix summarizes the syntax for file names and process identifiers. It is in four principal subsections.

The first subsection specifies reserved file names.

The second subsection describes the syntax that HP recommends for all new development. Any system procedure that has a name ending with an underscore (_) expects this syntax when you specify a file name or a process identifier as a parameter.

The third subsection describes the syntax that is supported in C-series software and is still supported in some system procedures in subsequent RVUs. Most of the procedure calls that support C-series file name syntax are marked in this manual as "superseded" and are listed in Appendix G, Superseded Guardian Procedure Calls and Their Replacements .

The fourth subsection describes the syntax for OSS pathnames.

Reserved File Names

Subvolume names and file names beginning with the letter "Z" are reserved. You should not choose such names in your application.

Syntax

This subsection summarizes the syntax for file names and process identifiers. It describes the syntax for four categories of file names along with file-name patterns and process handles. The four categories of file names are:

- Names that identify disk files
- Names that identify nondisk devices
- Names that identify unnamed processes
- Names that identify named processes

There is no distinction between external and internal file names. The system does not distinguish between uppercase and lowercase alphabetic characters in a file name. If all the optional left-hand parts of a file name are present, it is called a **fully qualified** file name; if any of the optional left-hand parts are missing, it is called a **partially qualified** file name.

Disk File Names

The syntax for a file name that identifies a disk file is:

```
[node.][[volume.]subvol.]file-id

or

[node.][volume.]temp-file-id
```

node

specifies the name of the node on which the file resides. A node name consists of a backslash (\) followed by one to seven alphanumeric characters; the first alphanumeric character must be a letter.

volume

specifies the name of the volume on which the file resides. A volume name consists of a dollar sign ($) followed by one to seven alphanumeric characters; the first alphanumeric character must be a letter.

subvol

specifies the name of the subvolume on which the file resides. A subvolume name has one to eight alphanumeric characters; the first character must be a letter.

file-id

specifies the file identifier (or name) of a permanent disk file. A permanent-file identifier has one to eight alphanumeric characters; the first character must be a letter.

temp-file-id

specifies the file identifier (or name) of a temporary disk file. A temporary-file identifier consists of a pound sign (#) followed by four to seven numeric characters. The operating system assigns file identifiers to temporary files.

Example

This is an example of a fully qualified disk file name:

```
\hdq.$mkt.reports.finance
```

Nondisk Device Names

The syntax for a file name that identifies a nondisk device is:

```
[node.]device-name[.qualifier]

or

[node.]ldev-number
```

node

specifies the name of the node on which the device resides. A node name consists of a backslash (\) followed by one to seven alphanumeric characters; the first alphanumeric character must be a letter.

device-name

specifies the name of a device. A device name consists of a dollar sign ($) followed by one to seven alphanumeric characters; the first alphanumeric character must be a letter.

qualifier

is an optional qualifier. It consists of a pound sign (#) followed by one to seven alphanumeric characters; the first alphanumeric character must be a letter.

ldev-number

specifies a logical device number. A logical device number is represented by a dollar sign ($) followed by a maximum of five digits. The logical device number 0 (represented “$0”) is reserved for the Event Management Service (EMS) collector process.

It is recommended that, wherever possible, the *device-name* form of a nondisk device name be used instead of the *ldev-number* form. This is especially true when referring to devices that are dynamically configured.

Examples

These are examples of file names that identify nondisk devices.

```
\ny.$ctlr.#term22
$s.#lp
$tape4
$10
```

Process File Names for Unnamed Processes

The syntax for a process file name that identifies an unnamed process is:

```
[node.]$:cpu:pin:seq-no
```

node

specifies the name of the node on which the process is running. A node name consists of a backslash (\) followed by one to seven alphanumeric characters; the first alphanumeric character must be a letter.

cpu

specifies the processor number of the processor in which the process is running. *cpu* is one or two digits representing a value in the range 0 through 15. A leading zero must be suppressed. A colon (:) separates the dollar sign ($) from *cpu*.

pin

specifies the process identification number of the process. *pin* is one to five digits representing a value in the range 0 through the maximum value allowed for the processor. Leading zeros must be suppressed. A colon separates *cpu* from *pin*.

seq-no

specifies the system-assigned sequence number of the process. *seq-no* has a maximum of 13 digits. Leading zeros must be suppressed. A colon separates *pin* from *seq-no*

Note. The sequence number is mandatory for unnamed processes. The sequence number cannot be removed from an unnamed process file name because a fatal error will result.

Example

These are examples of process file names that identify unnamed processes:

```
$:2:850:5237743650
\la.$:6:210:2876350012
\west.$:6:138:3547235420
```

Process File Names for Named Processes

The syntax for a process file name that identifies a named process is:

```
[node.]process-name[:seq-no][.qual-1[.qual-2]]
```

node

specifies the name of the node on which the process is running. A node name consists of a backslash (\) followed by one to seven alphanumeric characters; the first alphanumeric character must be a letter.

process-name

specifies the name of the process. A process name consists of a dollar sign ($) followed by one to five alphanumeric characters; the first alphanumeric character must be a letter.

seq-no

specifies the system-assigned sequence number of the process. *seq-no* has a maximum of 13 digits. Leading zeros must be suppressed. A colon (:) separates *process-name* from *seq-no*.

qual-1 and *qual-2*

are optional qualifiers. *qual-1* consists of a pound sign (#) followed by one to seven alphanumeric characters; the first alphanumeric character must be a letter.

qual-2 contains one to eight alphanumeric characters; the first character must be a letter.

Examples

These are examples of process file names that identify named processes.

```
\sw.$zab2:4300411433
$zsvr
\sf.$app2.#a001.z1
```

Process Descriptors

A process descriptor is a form of process file name that always includes the *node* and *seq-no* sections of the name; when identifying a named process, it never includes the optional qualifiers *qual-1* or *qual-2*. Guardian 90 procedures always return a process descriptor as the external-form representation of a process or process pair.

These are examples of process descriptors:

```
\node5.$zproc:1622091078
\east.$:5:131:436612
```

File-Name Patterns

A file-name pattern resembles a file name but designates a set of entities (that is, a set of disk files, devices, processes, or systems) through the use of pattern-matching characters. The pattern-matching characters are:

*	An asterisk matches zero or more letters, digits, dollar signs, pound signs, or a combination of these.
?	A question mark matches exactly one letter, digit, dollar sign, or pound sign.

The syntax for a file-name pattern is:

```
\pattern

or

[\pattern.]$pattern[.pattern[.pattern]]

or

[\pattern.][[pattern.]pattern.]pattern
```

pattern

consists of one or more characters. Allowable characters are letters, digits, pound signs (#), asterisks (*), and question marks (?). The maximum length of a *pattern* is twice that of the corresponding portion of a file name. (For example, 16 characters is allowed for a *pattern* that corresponds to a subvolume portion. This allows you to interleave multiple asterisks with a set of fixed characters.)

The all-numeric portions of a file name (that is, the *seq-no*, *cpu*, and *pin* portions of process file names) cannot be represented by *pattern*.

This syntax allows combinations of characters that are not permitted in file names, such as the use of pound signs anywhere in any portion. However, using such a combination of characters means that the pattern cannot designate any entity.

Note that the dollar sign is allowed only in the second form of the file-name pattern, as shown in the diagram above. The presence or absence of the dollar sign determines

whether the system interprets the file-name pattern according to the rules of the second or third form. As a result, the file-name pattern “$*.*” cannot match a permanent disk file name, but “*.*” can match a permanent disk file name (in the form *subvol.file-id*).

Examples

`*z*` matches all files in the current subvolume that have names (file IDs) containing the letter "z."

`$TERM??` matches all devices on the current node that have 7-character names starting with “$TERM,” such as “$TERM12.”

`P*.*` matches all disk files on the current volume that are in subvolumes whose names begin with the letter “P.”

Process Handles

A process handle is a 10-word structure that identifies a single named or unnamed process.

△ **Caution.** The format of a process handle is defined by HP and is subject to change in future RVUs. Applications should not try to extract information (such as processor or PIN) from a process handle except by using a system procedure such as PROCESSHANDLE_DECOMPOSE_.

A process handle contains this information about a process:

- The PIN, which identifies the process within a processor.
- The processor number, which identifies the processor in which the process is running.
- The node number, which identifies the node within a network.
- The sequence (or verifier) number, which allows the system to uniquely identify a process over its lifetime.
- The process pair index, which allows the system to locate the other member of a named process pair and to look up the process’s name.
- The type field, which indicates characteristics of the process (for example, whether the process is named or unnamed).

A process handle that contains -1 in each word is called a **null** process handle.

C-Series Syntax

This subsection summarizes the file name syntax that is supported in C-series RVUs and is still supported in the system procedures carried over from the C series. It describes both the external and internal forms of disk file names and nondisk file names. It also includes the syntax for process file names and process IDs.

Most of the procedure calls that support C-series file name syntax are marked in this manual as “superseded” and are listed in Appendix G.

External File Names

The external form of a file name is typically used when the user specifies a file name to a command interpreter or when the system displays a file name to the user. Both disk files and nondisk files can be represented by external file names.

An external file name consists of one or more parts, where adjacent parts are separated by a period. The system does not distinguish between uppercase and lowercase alphabetic characters in a file name. If all the optional left-hand parts of an external file name are present, it is called a **fully qualified** file name; if any of the optional left-hand parts are missing, it is called a **partially qualified** file name.

Disk File Names

The syntax for the external form of a disk file name is:

```
[system.][volume.][subvol.]file-id

or

[system.][volume.]temp-file-id
```

system

specifies the name of the system on which the file resides. A system name consists of a backslash (\) followed by one to seven alphanumeric characters; the first alphanumeric character must be a letter.

When *system* is included in a file name, it is called a **network** file name; when *system* is not included, it is called a **local** file name.

volume

specifies the name of the volume on which the file resides. A volume name consists of a dollar sign ($) followed by one to seven alphanumeric characters (one to six if you also specify *system*); the first alphanumeric character must be a letter.

subvol

specifies the name of the subvolume on which the file resides. A subvolume name has one to eight alphanumeric characters; the first character must be a letter.

file-id

specifies the file identifier (or name) of a permanent disk file. A permanent-file identifier has one to eight alphanumeric characters; the first character must be a letter.

temp-file-id

specifies the file identifier (or name) of a temporary disk file. A temporary-file identifier consists of a pound sign (#) followed by four digits. The operating system assigns file identifiers to temporary files.

Thsi is an example of a fully qualified external disk file name:

```
\hdq.$mkt.reports.finance
```

Nondisk File Names

A nondisk file name can identify a device or a process. The syntax for the external form of a nondisk file name is:

```
[system.]{ process-name }[.qual-1[.qual-2]]
         { device-name  }
         { ldev-number  }
```

system

specifies the name of the system on which the process or device resides. A system name consists of a backslash (\) followed by one to seven alphanumeric characters; the first alphanumeric character must be a letter.

When *system* is included in a file name, it is called a **network** file name; when *system* is not included, it is called a **local** file name.

process-name

specifies the name of a process. A process name consists of a dollar sign ($) followed by one to five alphanumeric characters (one to four if you also specify system); the first alphanumeric character must be a letter.

device-name

specifies the name of a device. A device name consists of a dollar sign ($) followed by one to seven alphanumeric characters (one to six if you also specify *system*); the first alphanumeric character must be a letter.

ldev-number

specifies a logical device number. A logical device number consists of a dollar sign ($) followed by one to four digits. The logical device number 0 (represented “$0”) is reserved for the Event Management Service (EMS) collector process.

qual-1 and *qual-2*

are optional qualifiers. *qual-2* cannot be used in combination with *device-name*; neither qualifier can be used in combination with *ldev-number*.

qual-1 consists of a pound sign (#) followed by one to seven alphanumeric characters; the first alphanumeric character must be a letter.

qual-2 contains one to eight alphanumeric characters; the first character must be a letter.

These are examples of external nondisk file names:

```
\sw.$proc.#out.default
\sw.$drvr.#term
$s.#lp
$tape4
$10
```

Internal File Names

An internal file name is a 12-word array in which the different file name parts begin at fixed locations in the array. The internal form of a file name is typically used within the system, as when a file name is passed between an application process and the operating system.

Except where noted, italicized syntax elements in these diagrams have the same definitions as they do for external file names.

Local File Names

The internal form of a local file name is as follows:

To access a permanent disk file, use

[0:3] *volume* (blank fill)
[4:7] *subvol* (blank fill)
[8:11] *file-id* (blank fill)

To access a temporary disk file, use

[0:3] *volume* (blank fill)
[4:11] *temp-file-id* (blank fill)

To access a nondisk device, use

[0:3] *device-name* or
ldev-number (blank fill)

[4:11][*qual-1*] (with *device-name* only) (blank fill)

To access $RECEIVE, use

[0:11]"$RECEIVE" (blank fill)

To access another process, if it is named, use

[0:3]*process-name* (blank fill)
[4:7][*qual-1*] (blank fill)
[8:11][*qual-2*] (blank fill)

Network File Names

The internal form of a network file name is:

[0].<0:7>	"\"
[0].<8:15>	System number (0 through 254)
[1:3]	*volume* (up to six characters), *device-name* (up to six characters), or *process-name* (up to four characters); no leading dollar sign (blank fill)
[4:11]	Same as local file name

Process File Names

A process file name is a 12-word array that uniquely identifies a process. There are three forms of the process file name: the timestamp form, local name form, and network form. Note that these forms cannot be used to designate a process that has a PIN greater than 255.

Timestamp Form of Process File Name

The timestamp form of the process file name is:

[0].<0:1>	2
[0].<2:7>	Reserved
[0].<8:15>	System number (0 through 254)
[1:2]	Low-order 32 bits of creation timestamp
[3].<0:3>	Reserved
[3].<4:7>	Processor in which the process resides
[3].<8:15>	PIN assigned by the system to identify the process in the processor
[4:11]	Blank-filled

Local Name Form of Process File Name

The local name form of the process file name is:

[0:2]	*process-name* (up to five characters plus leading "$") (blank fill)
[3].<0:3>	Reserved

[3].<4:7>	Processor in which the process resides; optional
[3].<8:15>	PIN assigned by the system to identify the process in the processor; optional
[4:7]	[*qual-1*] (blank fill)
[8:11]	[*qual-2*] (blank fill)

Network Form of Process File Name

The network form of a process file name is:

[0].<0:7>	"\"
[0].<8:15>	System number (0 through 254)
[1:2]	*process-name* (up to four characters; no leading "$") (blank fill)
[3].<4:7>	Processor in which the process resides; optional
[3].<8:15>	PIN assigned by the system to identify the process in the processor; optional
[4:7]	[*qual-1*] (blank fill)
[8:11]	[*qual-2*] (blank fill)

Process IDs

A process ID is a four-word array that uniquely identifies a process within a system. There are three forms of the process ID: the timestamp form, local name form, and network form. These forms of the process ID are identical to the first four words of the equivalent forms of the process file name, except that the processor and PIN fields are not optional (seeProcess File Names on page D-11). In other words, for any form of the process file name, words [0:3] of the process file name are identical to the process ID.

The process ID is sometimes called a CRTPID.

OSS Pathname Syntax

OSS pathnames can be up to PATH_MAX characters long, including a null termination character. PATH_MAX is a symbolic constant defined in the limitsh header file.

The syntax for an OSS pathname is:

```
[[/][directory/]...]filename
```

/

specifies the root directory when it appears at the beginning of a pathname. Otherwise, it separates directory names and filenames.

directory

specifies the name of a directory. All characters are valid except slash (/) and the ASCII NULL character. A hyphen (-) cannot be the first character of a directory

name. The maximum length is NAME_MAX characters, as defined in the limitsh header file (this value is 248). This directory names have special meaning:

. is the OSS current working directory

. . is the parent directory of the OSS current working directory

G

always appears under the root directory and identifies files in the Guardian file system.

E

always appears under the root directory and identifies files visible through Expand.

nodename

specifies the name of the node without a backslash (\ ) and it always resides after the E directory.

vol

specifies the name of the volume without a $ and it always resides after the G directory.

subvol

specifies the name of the subvolume and always resides after volume name.

filename

specifies the name of the file. All characters are valid except slash (/) and the ASCII NULL character. A hyphen (-) cannot be the first character of a filename. The maximum length is NAME_MAX characters, as defined in the limitsh header file (this value is 248).

Examples

OSS pathnames can be absolute or relative. Absolute pathnames begin with a slash (/), which indicates the root directory. This is an example of an absolute OSS pathname:

```
/usr/ccomp/prog1.c
```

Relative pathnames (which do not begin with a slash) are relative to the OSS current working directory. Examples of relative OSS pathnames follow:

refman/ch1	Refers to a file (ch1) in a subdirectory (refman) of the current working directory.
./refman/ch1	Refers to the same file as the previous example.

`../yourfiles/oldmail` Refers to a file (`oldmail`) in a subdirectory (`yourfiles`) of the parent directory of the current working directory.

`/E/forty/usr/ccom/prog1.c` Refers to a OSS file name (`prog1.c`) in a subdirectory (`ccom`) in a subdirectory (`usr`) in a subdirectory (`forty`) after the `E` root directory.

`/E/forty/G/books/donl/text180` Refers to a Guardian file name (`text180`) in a subvolume (`donl`) in a volume (`books`) on the HP node (`forty`).

For details on OSS pathnames and the OSS file system, see the *Open System Services Programmer's Guide* and the `filename(5)` reference page either online or in the *Open System Services System Calls Reference Manual*.

E DEFINEs

This appendix describes DEFINEs and the attributes of the different classes of DEFINEs. For information about using DEFINEs programmatically, see the *Guardian Programmer's Guide*. For information about using DEFINEs interactively with a TACL process, see the *Guardian User's Guide*.

What Is a DEFINE?

A DEFINE is a named set of attributes and associated values. In a DEFINE (as with an ASSIGN command) you can specify information that is to be communicated to processes you start. The operating system (file system or I/O processes) usually process DEFINES, while application programs or run-time libraries process ASSIGNS.

There are eight classes of DEFINEs. You can use the classes of DEFINEs in these ways:

- Use a CLASS CATALOG DEFINE to specify a substitute name for an SQL catalog name.
- Use a CLASS DEFAULTS DEFINE to specify process defaults, such as default volume and subvolume.
- Use a CLASS MAP DEFINE to specify a substitute name for a file name.
- Use a CLASS SEARCH DEFINE to specify a list of subvolumes for resolving file names with a search list.
- Use CLASS SORT and SUBSORT DEFINEs to specify defaults for the FASTSORT utility and for parallel sorts running under FASTSORT.
- Use a CLASS SPOOL DEFINE to specify the attributes of a spooler job.
- Use a CLASS TAPE DEFINE to specify the attributes of a file on labeled tape.
- Use a CLASS TAPECATALOG DEFINE to use the tape cataloging facilities of the DSM/TC product.

DEFINE Names

A DEFINE is identified by a name, which you specify when creating the DEFINE. The name must conform to these rules:

- The name must be 2 to 24 characters long.
- The first character must be an equal sign (=).
- The second character must be a letter. (DEFINE names whose second character is an underscore are reserved for use by HP.)
- The remaining characters can be letters, numbers, hyphens, underscores, or circumflexes(^).

- When specified as the value of a procedure parameter that has a fixed length of 24 characters, a DEFINE name must be left-justified in the DEFINE name buffer and padded on the right with blanks.

Uppercase and lowercase letters in a DEFINE name are equivalent. For example, the name =MY^DEFINE is equivalent to =My^Define.

DEFINE Attributes

A set of attributes is associated with each DEFINE. One attribute that is associated with every DEFINE is the CLASS attribute. The CLASS attribute determines which other attributes can be associated with the DEFINE.

Each attribute has:

- An attribute name that you cannot change.
- A data type that determines the kind of value that you can assign to the attribute.
- A value that you assign programmatically by a call to the DEFINESETATTR procedure, or interactively by the TACL SET DEFINE command. Some attributes have default values.

Attribute Data Types

When you assign a value to an attribute, you specify the value as a parameter to a procedure call. This parameter must be declared type STRING.

The string values that you can specify for a particular DEFINE attribute is determined by the data type of the DEFINE attribute. The available attribute data types are:

String	The attribute can contain a string of from 1 to a maximum of 512 ASCII characters, depending on the particular attribute.
Number	The attribute can contain an integer consisting of from 1 to a maximum of 18 digits, depending on the particular attribute. This integer can be preceded by a plus or minus sign and must not contain a decimal point. On output, the integer is left-justified with leading zeros suppressed.
Filename	The attribute can contain a file name. The file name can be fully or partially qualified. A partially qualified file name is expanded using the *default-names* value that you specify to the DEFINESETATTR procedure. On output, the file name is always fully qualified.
Subvolname	The attribute can contain a subvolume name. The subvolume name can be fully or partially qualified. A partially qualified subvolume name is expanded using the *default-names* value that you specify to the DEFINESETATTR procedure. On output, the subvolume name is always fully qualified, except when it is obtained from the VOLUME attribute of a CLASS DEFAULTS DEFINE.
Keyword	The attribute can contain one of a predefined set of keywords. These keywords are specific to the particular DEFINE attribute.

CLASS Attribute

All DEFINEs have a special attribute called the CLASS attribute. The CLASS attribute determines which other attributes are associated with the DEFINE.

The value of the CLASS attribute is a keyword; the CLASS attribute can be CATALOG, DEFAULTS, MAP, SEARCH, SORT, SPOOL, SUBSORT, TAPE, or TAPECATALOG. When assigning values to DEFINE attributes, you must assign one of these values to the CLASS attribute first. Assigning a value to the CLASS attribute causes default values to be assigned to other attributes in that DEFINE class.

The attributes of a particular DEFINE are distinct from attributes of other DEFINE classes, even when the attributes have the same names.

Available DEFINE Classes

The DEFINE classes that are currently available are described in these paragraphs.

CLASS CATALOG DEFINEs

A CLASS CATALOG DEFINE substitutes an SQL catalog name for the DEFINE name in a program.

The attribute of a CLASS CATALOG DEFINE is SUBVOL, which specifies the SQL catalog subvolume name to be substituted for the DEFINE name. For detailed information about the CLASS CATALOG DEFINE and its attributes, see the *Guardian User's Guide* and the *SQL/MP Reference Manual*.

CLASS DEFAULTS DEFINEs

A CLASS DEFAULTS DEFINE contains standard defaults such as the default volume and subvolume to be used by a process. For detailed information about the CLASS DEFAULTS DEFINE and its attributes, see the *Guardian User's Guide* and the *TACL Reference Manual*.

CLASS MAP DEFINEs

A CLASS MAP DEFINE allows you to substitute a logical DEFINE name for an actual file name in a program.

The attribute of a CLASS MAP DEFINE is FILE, which specifies the file name to be substituted for the DEFINE name. For detailed information about the CLASS MAP DEFINE and its attributes, see the *Guardian User's Guide* and the *TACL Reference Manual*.

CLASS SEARCH DEFINEs

A CLASS SEARCH DEFINE contains information to be used for resolving file names with a search list.

A CLASS SEARCH DEFINE has 21 attributes named SUBVOL0 through SUBVOL20 and another 21 attributes named RELSUBVOL0 through RELSUBVOL20. Each of these attributes takes the same form and is optional. The value of one attribute is either a single subvolume specification or a list of them enclosed in parentheses and separated by commas. A subvolume specification can be a fully or partially qualified subvolume name, or the name of a CLASS DEFAULTS DEFINE.

With the SUBVOLnn attributes, subvolume name resolution takes place when the attribute is added; with the RELSUBVOLnn attributes, subvolume name resolution takes place when the DEFINE is used. The search order for a CLASS SEARCH DEFINE is as follows:

SUBVOL0
RELSUBVOL0
SUBVOL1
RELSUBVOL1
. . .
SUBVOL20
RELSUBVOL20

If any attribute is a list, the search order is from left to right within the list.

CLASS SEARCH DEFINEs are used by the FILENAME_RESOLVE_ procedure. For detailed information about the CLASS SEARCH DEFINEs and their attributes, see the *Guardian Programmer's Guide*.

CLASS SORT DEFINEs

A CLASS SORT DEFINE passes information to the FASTSORT utility. All SORT attributes (other than CLASS) are optional.

FASTSORT always checks for the presence of a DEFINE named =_SORT_DEFAULTS. If this DEFINE exists and is of CLASS SORT, FASTSORT reads the attributes from it and uses them to set the sort parameters. =_SORT_DEFAULTS is reserved for use as the name for a default CLASS SORT DEFINE. For detailed information about the CLASS SORT DEFINE and its attributes, see the *FastSort Manual*.

CLASS SUBSORT DEFINEs

A CLASS SUBSORT DEFINE passes information that applies to parallel sorts running under the FASTSORT utility. The only required attribute (other than CLASS) is SCRATCH. For detailed information about the CLASS SUBSORT DEFINE and its attributes, see the *FastSort Manual*.

CLASS SPOOL DEFINEs

A CLASS SPOOL DEFINE passes information to the spooler collector process to assign values to spooler job attributes. For detailed information about the CLASS SPOOL DEFINE and its attributes, see the *Spooler Utilities Reference Manual* and the *Spooler Plus Utilities Reference Manual*.

CLASS TAPE DEFINEs

A CLASS TAPE DEFINE passes information to the tape process when using labeled magnetic tapes. One CLASS TAPE DEFINE must be used for each labeled tape file that is accessed by your application. CLASS TAPE DEFINEs are processed by the tape process and by the FILE_OPEN_ and OPEN procedures. For detailed information about the CLASS TAPE DEFINEs and their attributes, see the *Guardian Disk and Tape Utilities Reference Manual*.

CLASS TAPECATALOG DEFINEs

A CLASS TAPECATALOG DEFINE is used to invoke the services of the DSM/TC product. It is used in place of a CLASS TAPE DEFINE, adding several attributes for control of cataloging files that are read from and written to tape.

One CLASS TAPECATALOG DEFINE must be used for each labeled tape file that you want to read or write. CLASS TAPECATALOG DEFINEs are processed by the FILE_OPEN_ and OPEN procedures. The DSM/TC facility automatically selects tape volumes and catalogs the files written to tape by applications using CLASS TAPECATALOG DEFINEs. For detailed information about the CLASS TAPECATALOG DEFINEs and their attributes, see the *DSM/Tape Catalog User's Guide*.

F Formatter Edit Descriptors

This appendix describes edit descriptors, which are specified as input values to the FORMATCONVERT procedure.

Edit descriptors are of two types: those that specify the conversion of data values (repeatable) and those that do not (nonrepeatable). The effect of repeatable edit descriptors can be altered through the use of modifiers or decorations, which are enclosed in brackets ([ ]), preceding the edit descriptors to which they refer. Within a format, all edit descriptors except buffer control descriptors must be separated by commas. Buffer control descriptors have the dual function of edit descriptors and format separators, and need not be set off by commas.

Summary of Edit Descriptors

All the descriptors, modifiers, and decorations are summarized here and fully explained following this summary.

Summary of Nonrepeatable Edit Descriptors

The edit descriptors that are not associated with data items are of six subtypes:

- Tabulation

  T*n* Tab absolute to nth character position
  TR*n* Tab right
  TL*n* Tab left
  *n*X Tab right (same as TR)

- Literals

  Alphanumeric string enclosed in apostrophes (') or quotation marks (")

  Hollerith descriptor (*n*H followed by *n* characters)

- Scale factor specification

  P Implied decimal point in a number

- Optional plus control

  These descriptors provide control of the appearance of an optional plus sign for output formatting. They have no effect on input.

  S Do not supply a plus
  SP Supply a plus
  SS Do not supply a plus

- Blank interpretation control

  BN Blanks ignored (unless entire field is blank)
  BZ Blanks treated as zeros

- Buffer control

  / Terminate the current buffer, and then obtain a new one

  : Terminate formatting if no data elements remain

Summary of Repeatable Edit Descriptors

Repeatable edit descriptors direct the formatter to obtain the next data list element and perform a conversion between internal and external representation. They can be preceded by modifiers or decorations that alter the interpretation of the basic edit descriptor. Modifiers and decorations apply only to output conversion. They are allowed but ignored for input.

The repeatable edit descriptors are:

A	Alphanumeric (ASCII)
B	Binary (base 2) integer
D,E	Exponential form
F	Fixed form
G	General (E or F format depending on magnitude of data)
I	Integer (base *b*)
L	Logical
M	Mask formatting
O	Octal (base 8) integer
Z	Hexadecimal (base 16) integer

Summary of Modifiers

Modifiers are codes that are used to alter the results of the formatting prescribed by the edit descriptors to which they are attached. They are:

BN, BZ	Field blanking (if null, or zero)
FL	Fill-character specification
LJ, RJ	Left and right justification
OC	Overflow-character modifier
SS	Symbol substitution

Summary of Decorations

Decorations specify alphanumeric strings that can be added to a field either before basic formatting is begun or after it is finished. A decoration consists of one or more codes that specify the conditions under which the string is to be added (based on the value of the data element or the occurrence overflow of the external field):

M	Minus

N Null

O Overflow

P Plus

Z Zero

followed by a code that describes the position of the special editing:

A (absolute) At a specific character position within the field

F (floating) At the position the basic formatting finished

P (prior) At the position the basic formatting would have started

followed by the character string that is to be included in the field if the stated conditions are met.

Nonrepeatable Edit Descriptors

These descriptions show the form, function, and requirements for each of the nonrepeatable edit descriptors.

Tabulation Descriptors

The tabulation descriptors specify the position at which the next character is transmitted to or from the buffer. This allows portions of a buffer to be processed in an order other than strictly left to right, and permits processing of the same portion of a buffer more than once.

The forms of the tabulation descriptors are as follows (*n* is an unsigned integer constant):

T*n* TL*n* TR*n* *n*X

T*n* Indicates that the transmission of the next character to or from a buffer is to occur at the *n*th character position. The first character of the buffer is numbered 1.

TL*n* Indicates that the transmission of the next character to or from the buffer is to occur *n* positions to the left of the current position

TR*n* Indicates that the transmission of the next character to or from the buffer is to occur *n* positions to the right of the current position

*n*X Is identical to TR*n*

Each of these edit descriptors alters the current position but has no other effect.

The current position can be moved beyond the limits of the current buffer (that is, become less than or equal to zero, or greater than *bufferlen*) without an error resulting, provided that no attempt is made by a subsequent edit descriptor to transmit data to or from a position outside the current buffer. Tab descriptors cannot be used to advance to later buffers or to return to previous ones. These examples illustrate tabulation descriptors:

Data List Values

```
100
1000.49F
"HELLO"
```

	Format	Results
No tabs	I3,E12.4,A5	100 0.1000E+04HELLO /\ /\ /\ /\
X	I3,E12.4,1X,A5	100 0.1000E+04 HELLO /\ /\ /\ /\
TL	I3,E12.4,TL3,A5	100 0.1000EHELLO /\ /\ /\ /\
TR	I3,E12.4,TR5,A5	100 0.1000E+04 HELLO /\ /\ /\ /\
T	I3,E12.4,T3,A5	10HELLO1000E+04 /\ /\ /\ /\

The "/\" marker denotes the boundaries of the output field.

Literal Descriptors

Literal descriptors are alphanumeric strings in either form:

dc c c ... c d OR *n*H*c c c ... c*
1 2 3 *n* 1 2 3 *n*

- *d* either an apostrophe (') or a quotation mark ("); the same character must be used for both the opening and closing delimiters.
- *c* any ASCII character.
- *n* an unsigned nonzero integer constant specifying the number of characters in the string; n cannot exceed 255.

On input, a literal descriptor is treated as *n*X.

A literal edit descriptor causes the specified character string to be inserted in the current buffer beginning at the current position. It advances the current position *n* characters.

In a quoted literal form, if the character string to be represented contains the same character that is used as the delimiter, two consecutive characters are used to distinguish the data character from the delimiter; for example:

To represent:	Use:		
`can't`	`'can''t'`	or	`"can't"`
`"can't"`	`'"can''t"'`	or	`"""can't"""`

In the Hollerith constant form, the number of characters in the string (including blanks) must be exactly equal to the number preceding the letter H. There are no delimiter characters, so the characters are supplied exactly as they should appear in the buffer; for example:

To represent:	Use:
`can't`	`5Hcan't`

Scale-Factor Descriptor (P)

The form of a scale-factor descriptor is:

*n*P

n = optionally signed integer in the range of -128 to 127.

The value of the scale factor is zero at the beginning of execution of the FORMATDATA procedure. Any scale-factor specification remains in effect until a subsequent scale specification is processed. The scale factor applies to the D, E, F, and G edit descriptors, affecting them in this manner:

1. On input, with D, E, F, and G edit descriptors (provided no exponent exists in the external field), the scale-factor effect is that the externally represented number equals the internally represented number multiplied by 10**_n_.
2. On input, with D, E, F, and G edit descriptors, the scale factor has no effect if there is an exponent in the external field.
3. On output, with D and E edit descriptors, the mantissa of the quantity to be produced is multiplied by 10**_n_, and the exponent is reduced by _n_.
4. On output, with the F edit descriptor, the scale-factor effect is that the externally represented number equals the internally represented number multiplied by 10**_n_.
5. On output, with the G edit descriptor, the effect of the scale factor is suspended unless the magnitude of the data to be processed is outside the range that permits the use of an F edit descriptor. If the use of the E edit descriptor is required, the scale factor has the same effect as with the E output processing.

Optional Plus Descriptors (S, SP, SS)

Optional plus descriptors can be used to control whether optional plus characters appear in numeric output fields. In the absence of explicit control, the formatter does not produce any optional plus characters.

The forms of the optional plus descriptors are:

S SP SS

These descriptors have no effect upon input.

If the S descriptor is encountered in the format, the formatter does not produce a plus in any subsequent position that normally contains an optional plus.

If the SP descriptor is encountered in the format, the formatter produces a plus in any subsequent position that normally contains an optional plus.

The SS descriptor is the same as S (above).

An optional plus is any plus except those appearing in an exponent.

Blank Interpretation Descriptors (BN, BZ)

The blank interpretation descriptors have this form:

BN BZ

These descriptors have no effect on output.

The BN and BZ descriptors can be used to specify the interpretation of blanks, other than leading blanks, in numeric input fields. At the beginning of execution of the FORMATDATA procedure, nonleading blank characters are ignored.

If a BZ descriptor is encountered in a format, all nonleading blank characters in succeeding numeric input fields are treated as zeros.

If a BN descriptor is encountered in a format, all blank characters in succeeding numeric input fields are ignored. The effect of ignoring blanks is to treat the input field as if all blanks had been removed, the remaining portion of the field right-justified, and the blanks reinserted as leading blanks. However, a field of all blanks has the value zero.

The BN and BZ descriptors affect the B, D, E, F, G, I O, and Z edit descriptors only.

Buffer Control Descriptors (/, :)

There are two edit descriptors used for buffer control:

/ indicates the end of data list item transfer on the current buffer and obtains the next buffer. The current position is moved to 1 in preparation for processing the next buffer.

: indicates termination of the formatting provided there are no remaining data elements.

- To clarify, the operation of the slash (/) is as follows for any positive integer *n*:
  - If *n* consecutive slashes appear at the end of a format, this causes *n* buffers to be skipped.
  - If *n* consecutive slashes appear within the format, this causes *n*-1 buffers to be skipped.
- The colon (:) is used to conditionally terminate the formatting. If there are additional data list items, the colon has no effect. The colon can be of use when data items are preceded by labels, as in this example:

```
10(' NUMBER ',I1,:/)
```

This group of edit descriptors is preceded by a repeat factor that specifies the formatting of ten data items, each one to be preceded by the label NUMBER. If there are fewer than ten data items in the data list, formatting terminates immediately after the last value is processed. If the colon is not present, formatting continues until the I edit descriptor is encountered for the fourth time. This means the fourth label is added before the formatting is terminated.

These example illustrates this usage:

Data Items:

```
1
2
3
```

Format:

With colon	**Without colon**
`10('NUMBER ',I1,:/)`	`10('NUMBER ',I1,/)`

Results:

With colon	**Without colon**
|NUMBER 1|	|NUMBER 1|
|NUMBER 2|	|NUMBER 2|
|NUMBER 3|	|NUMBER 3|
	|NUMBER |

The “|” character is used to denote the boundaries of the output field.

Repeatable Edit Descriptors

These descriptions give the form, function, and requirements for each of the edit descriptors that specify formatting of data fields. These edit descriptors can be preceded by an unsigned integer repeat factor to specify identical formatting for a number of values in the data list.

These descriptions of the operation of repeatable edit descriptors apply when no decorations or modifiers are present.

The A Edit Descriptor

The A edit descriptor is used to move characters between the buffer and the data element without conversion. This is normally used with ASCII data. The A edit descriptor has one of these forms:

A*w* OR A

w an unsigned integer constant that specifies the width, in characters, of the field and must not exceed 255. The field processed is the next *w* characters starting at the current position.

If *w* is not present, the field width is equal to the actual number of bytes in the associated data element, but cannot exceed 255. Values over 255 are reduced to 255.

After the field is processed, the current position is advanced by w characters.

On output, the operation of the A edit descriptor is as follows:

1. The number of characters specified by *w*, or the number of characters in the data element, whichever is less, is moved to the external field. The transfer starts at the left character of both the data element and the external field unless an RJ modifier is affecting the descriptor, in which case the transferring of characters begins with the right character of each.
2. If *w* is less than the number of characters in the data element, the field overflow condition is set.
3. If *w* is greater than the number of characters in the data element, the remaining characters in the external field are filled with spaces (unless another fill character is specified by the FL modifier).

It is not mandatory that the data element be of type character. For example, an INTEGER(16) element containing the octal value %015536 corresponds to the ASCII

characters “ESC” and “^”, which can be output to an ADM-2 terminal using an A2 descriptor to control a blinking field on the screen. For example:

Format	Data Value	External Field
A	'WORD'	|WORD|
A4	'WORD'	|WORD|
A3	'WORD'	|WOR| (overflow set)
[RJ]A3	'WORD'	|ORD| (overflow set)
A5	'WORD'	|WORD |
[RJ]A5	'WORD'	| WORD|
A	%044111	|HI|

In the last example, the data value was stored in a 2-byte INTEGER.

The “|” character is used to denote the boundaries of the output field.

On input, the operation of the A*w* edit descriptor is as follows:

1. The number of characters specified by *w*, or the number of characters contained in the data element specified by *n*, whichever is less, is moved from the external field to the data element. The transfer begins at the left character of both the data element and the external field.
2. If *w* is less *n*, the data element is filled with (*n*-*w*) blanks on the right.
3. If *w* is greater than *n*, the leftmost *n* characters of the field are stored in the data element.

These examples illustrate these considerations:

External Field	Format	Data Item Length	Data Element Value
|HELLO|	A5	5 characters	'HELLO'
|HELLO|	A3	3 characters	'HEL'
|HELLO|	A6	6 characters	'HELLO '
|HELLO|	A5	6 characters	'HELLO '
|HELLO|	A5	3 characters	'HEL'

The “|” character is used to denote the boundaries of the input field.

The B Edit Descriptor

The binary edit descriptor is used to display or interpret data values in binary (base 2) integer form.

The B edit descriptor has these forms:

B*w* OR B*w.m*

w an unsigned integer constant that defines the total field width and cannot exceed 255. The field processed is the *w* characters starting at the current position. After the field is processed, the current position is advanced by *w* characters.

m an unsigned integer constant that defines the number of digits that must be present on output.

The B edit descriptor is used in the same manner as the I edit descriptor where the number base (*b*) is 2, except that the B edit descriptor always treats the internal data value as unsigned. (See The I Edit Descriptor on page F-14.) For example, if the data item is an INT(16) in TAL, these conversions take place:

Format	Internal Value	Result
B16	5	| 101|
B16.6	3	| 000011|
B16.6	-5	|1111111111111101|

The "|" character is used to denote the boundaries of the output field.

The D Edit Descriptor

The exponential edit descriptor is used to display or interpret data in floating-point form, usually used when data values have extremely large or extremely small magnitude. The D edit descriptor is of the form:

D*w.d*

This descriptor is identical to the E*w.d* descriptor.

This edit descriptor is used in the same manner as the E edit descriptor (below).

The E Edit Descriptor

The exponential edit descriptor is used to display or interpret data in floating-point form. It is usually used when data values have extremely large or extremely small magnitude.

The E edit descriptor has one of these forms:

E*w.d* OR E*w.d*E*e*

w an unsigned integer constant that defines the total field width (including the exponent) and cannot exceed 255. The field processed is the *w* characters starting at the current position. After the field is processed, the current position is advanced by *w* characters.

d an unsigned integer constant that defines the number of digits that are to appear to the right of the decimal point in the external field.

e an unsigned integer constant that defines the number of digits in the exponent. If E*w.d* is used, *e* takes the value 2.

The input field consists of an optional sign, followed by a string of digits optionally containing a decimal point. A decimal point appearing in the input field overrides the portion of the descriptor that specifies the decimal point location. However, if you omit the decimal point, the rightmost *d* digits of the string, with leading zeros assumed if necessary, are interpreted as the fractional part of the value represented. The string of digits can be of any length. Those beyond the limit of precision of the internal representation are ignored. The basic form can be followed by an exponent in one of these forms:

- Signed integer constant
- E followed by zero or more blanks, followed by an optionally signed integer constant
- D followed by zero or more blanks, followed by an optionally signed integer constant

An exponent containing a D is processed identically to an exponent containing an E.

On output, the field (for a scale factor of zero) appears in this form:

```
{[+]} [0].n n  … n       E {+}  e e …e
{ - }       1 2      d       {-}   1 2    e
```

`{[+]}`	Indicates an optional plus or a minus
`{ - }`	
n n ... n	Are the *d* most significant digits of the value of the data
1 2 *d*	after rounding
E	Signals the start of the decimal exponent
`{+}`	Indicates that a plus or minus is required
`{-}`	
e e ... e	Are the *e* most significant digits of the exponent
1 2 *e*	

The sign in the exponent is always displayed. If the exponent is zero, a plus sign is used.

If the data is negative, the minus sign is always displayed. If the data is positive (or zero), the display of the plus sign is dependent on the last optional plus descriptor processed.

The zero preceding the decimal point is normally displayed, but can be omitted to prevent field overflow. Decimal normalization is controlled by the scale factor established by the most recently interpreted *n*P edit descriptor. If -*d* < *n* <= 0, the output value has |*n*| leading zeros, and (*d*-|*n*|) significant digits follow the decimal point; if 0 < *n* < *d*+2, the output value has n significant digits to the left of the decimal point and *d*-*n*+1 digits to the right. If the number of characters produced exceeds the

field width or if an exponent exceeds its specified length using the E*w*.*d*E*e* field descriptor, the entire field of width *w* is filled with asterisks. However, if the field width is not exceeded when optional characters are omitted, the field is displayed without the optional characters.

Because all characters in the output field are included in the field width, *w* must be large enough to accommodate the exponent, the decimal point, and all digits and the algebraic sign of the base number.

These examples illustrate output:

Format	Data Value	Result
E12.3	8.76543×10^{-6}	| 0.877E-05|
E12.3	-0.55555	| -0.556E+00|
E12.3	123.4567	| 0.123E+03|
E12.6E1	3.14159	| 0.314159E+1|

The "|" character is used to denote the boundaries of the output field.

Note. To use the E edit descriptor for output, floating-point firmware is required.

These examples illustrate input:

External Field	Format	Data Element Value
| 0.100E+03|	E12.3	100
| 100.05 |	E12.5	100.05
| 12345|	E12.3	12.345

The "|" character is used to denote the boundaries of the output field.

The F Edit Descriptor

The fixed-format edit descriptor is used to display or interpret data in fixed point form.

The F edit descriptor has these forms:

F*w*.*d* OR F*w*.*d*.*m*

w an unsigned integer constant that defines the total field width and cannot exceed 255. The field processed is the *w* characters starting at the current position. After the field is processed, the current position is advanced by *w* characters.

d an unsigned integer constant that defines the number of digits that are to appear to the right of the decimal point in the external field.

m an unsigned integer constant that defines the number of digits that must be present to the left of the decimal point on output.

On input, the F*w*.*d* edit descriptor is the same as the E*w*.*d* edit descriptor.

The output field consists of blanks if necessary, followed by a minus if the internal value is negative or an optional plus otherwise. This is followed by a string of digits that contains a decimal point and represents the magnitude of the internal value, as modified by the established scale factor and rounded to the d fractional digits. If the magnitude of the value in the output field is less than one, there are no leading zeros except for an optional zero immediately to the left of the decimal point. The optional zero must appear if there would otherwise be no digits in the output field. If the F*w.d.m* form is used, leading zeros are supplied if needed to satisfy the requirement of *m* digits to the left of the decimal point. For example:

Format	Data Value	Result
F10.4	123.4567	| 123.4567|
F10.4	0.000123	| 0.0001|
F10.4.3	-4.56789	| -004.5679|

The "|" character is used to denote the boundaries of the output field.

The G Edit Descriptor

The general format edit descriptor can be used in place of either the E or the F edit descriptor, since it has a combination of the capabilities of both.

The G edit descriptor has either of the forms:

G*w.d* OR G*w.d*E*e*

- *w* an unsigned integer constant that defines the total field width and cannot exceed 255. The field processed is the *w* characters starting at the current position. After the field is processed, the current position is advanced by *w* characters.
- *d* an unsigned integer constant that defines the number of digits that are to appear to the right of the decimal point in the external field.
- *e* an unsigned integer constant that defines the number of digits in the exponent, if one is present.

On input, the G edit descriptor is the same as the E edit descriptor. The method of representation in the output field depends on the magnitude of the data being processed, as follows:

Magnitude of Data		
Not Less Than	**Less Than**	**Equivalent Conversion Effected**
	0.1	E*w.d* or E*w.d*E*e*
0.1	1.0	F(*w-n*).*d*,*n*(' ')
1.0	10.0	F(*w-n*).(*d*-1),*n*(' ')

Magnitude of Data		
Not Less Than	**Less Than**	**Equivalent Conversion Effected**
10.0	100.0	F(*w*-*n*).(*d*-2),*n*(' ')
.	.	.
.	.	.
.	.	.
10 ** (*d*-2)	10 ** (*d*-1)	F(*w*-*n*).1,*n*(' ')
10 ** (*d*-1)	10 ** *d*	F(*w*-*n*).0,*n*(' ')
10 ** *d*		E*w*.*d* or E*w*.*d*E*e*

The value of *n* is 4 for G*w*.*d* format and (*e*+2) for G*w*.*d*E*e* format. The *n*(' ') used in the above example indicates *n*th number of blanks. If the F form is chosen, then the scale factor is ignored. This comparison between F formatting and G formatting is given by way of illustration:

	F	G
Value	**F13.6 Conversion**	**G13.6 Conversion**
.01234567	| 0.012346|	| 0.123457E-01|
.12345678	| 0.123457|	| 0.123457 |
1.23456789	| 1.234568|	| 1.23457 |
12.34567890	| 12.345679|	| 12.3457 |
123.45678900	| 123.456789|	| 123.457 |
1234.56789000	| 1234.567890|	| 1234.57 |
12345.67890000	| 12345.678900|	| 12345.7 |
123456.78900000	|123456.789000|	| 123457. |
1234567.89000000	|**********|	| 0.123457E+07|

When an overflow condition occurs in a numeric field, the field is filled with asterisks (in the absence of any specification to the contrary by an overflow decoration), as shown above.

The "|" character is used to denote the boundaries of the output field.

Note. To use the G edit descriptor for output, floating-point firmware is required.

The I Edit Descriptor

The integer edit descriptor is used to display or interpret data values in an integer form.

The I edit descriptor has these forms:

I*w* OR I*w.m* OR I*w.m.b*

w an unsigned integer constant that defines the total field width and cannot exceed 255. The field processed is the *w* characters starting at the current position. After the field is processed, the current position is advanced by *w* characters.

m an unsigned integer constant that defines the number of digits that must be present on output.

b an unsigned integer constant that defines the number base of the external data and cannot be less than 2 or greater than 16.

On output, the I*w* edit descriptor causes the external field to consist of zero or more leading blanks (followed by a minus if the value of the internal data is negative, or an optional plus otherwise), followed by the magnitude of the internal value in the form of an unsigned integer constant without leading zeros. An integer constant always consists of at least one digit. If the number of characters produced exceeds the value of *w*, the entire field of width *w* is filled with asterisks.

The output from an I*w.m* edit descriptor is the same as that from the I*w* edit descriptor, except that the unsigned integer constant consists of at least *m* digits and, if necessary, has leading zeros. The value of *m* must not exceed the value of *w*. If *m* is zero and the internal data is zero, the output field consists only of blank characters, regardless of the sign control in effect.

The output from an I*w.m.b* edit descriptor is the same as that from the I*w.m* edit descriptor, except that the unsigned integer constant is represented in the number base *b*. With the I*w* edit and I*w.m* edit descriptors, the output is treated as if *b* were present and equal to 10. For example:

Format	Data Value	Result
I7	100	| 100|
I7.2	-1	| -01|
I7.6	100	| 000100|
I7.6	-1	|-000001|
I7.6.8	28	| 000034|
I7.1.2	-5	| -101|

The “|” character is used to denote the boundaries of the output field.

On input, the I*w.m* edit descriptor and the I*w.m.b* edit descriptor are treated identically to the I*w* edit descriptor. These edit descriptors indicate that the field to be edited occupies *w* positions. In the input field, the character string must be in the form of an optionally signed integer constant consisting only of base *b* digits, except for the interpretation blanks. Leading blanks on input are not significant, and the interpretation

of any other blanks is determined by blank control descriptors (BN and BZ). For example:

External Field	Format	Data Element Value
`|  100|`	`I7`	`100`
`|  -01|`	`I7`	`-1`
`| 1  |`	`I7`	`1`
`|  1  |`	`BZ,I7`	`1000`
`| 1 2 |`	`BZ,I7`	`10200`
`| 1 2 |`	`BN,I7`	`12`

The "|" character is used to denote the boundaries of the output field.

The L Edit Descriptor

The logical edit descriptor is used to display or interpret data in logical form. The L edit descriptor has the form:

L*w*

w an unsigned integer constant that defines the total field width and cannot exceed 255. The field processed is the *w* characters starting at the current position. After the field is processed, the current position is advanced by *w* characters.

On output, the L edit descriptor causes the associated data element to be evaluated in a logical context, and a single character is inserted right-justified in the output field. If the data value is null, the character is blank. If the data value is zero, the character is F; for all other cases, the character is T. For example:

Format	Data Value	Result
`L2`	`-1`	`| T|`
`L2`	`15769`	`| T|`
`L2`	`0`	`| F|`

The "|" character is used to denote the boundaries of the output field.

The input field consists of optional blanks, optionally followed by a decimal point, followed by an uppercase T for true (logical value -1) or an uppercase F for false (logical value 0). The T or F can be followed by additional characters in the field. The logical constants .TRUE. and .FALSE. are acceptable input forms; for example:

External Field	Format	Data Element Value
`|  T|`	`L7`	`-1`
`|  F|`	`L7`	`0`
`| .TRUE.|`	`L7`	`-1`

External Field	Format	Data Element Value
|.FALSE.|	L7	0
|TUGBOAT|	L7	-1
|FARLEY |	L7	0

The “|” character is used to denote the boundaries of the output field.

The M Edit Descriptor

The mask formatting edit descriptor edits either alphanumeric or numeric data according to an editing pattern or mask. Special characters within the mask indicate where digits in the data are to be displayed; other characters are duplicated in the output field as they appear in the mask. The M edit descriptor has the form:

M'*mask*'

mask a character string; *mask* can be enclosed in apostrophes ('), quotation marks ("), or less-than and greater-than symbols (<>). The string supplied must not exceed 255 characters.

The M edit descriptor is not allowed for input.

Characters in a mask that have special functions are:

Z Digit selector

9 Digit selector

V Decimal alignment character

. Decimal alignment character

The field width *w* is determined by the number of characters, including spaces but excluding Vs, between the mask delimiters. The field processed is *w* characters starting at the current position. After the field is processed, the current position is advanced by *w* characters. Except for the decimal point alignment character, V, each character in the mask either defines a character position in the field or is directly inserted in the field.

The M edit descriptor causes numeric data elements to be rounded to the number of positions specified by the mask. String data elements are processed directly. Each digit or character of a data element is transferred to the result field in the next available character position that corresponds to a digit selector in the mask. If the digit selector is a 9, it causes the corresponding data digit to be transferred to the output field. The digit selector Z causes a nonzero, or embedded zero, digit to be transferred to the field, but inserts blanks in place of leading or trailing zeros. Character positions must be allocated, by Z digit selectors, within the mask to provide for the inclusion of any minus signs or decoration character strings. A decimal point in the mask can be used for decimal point alignment of the external field. The letter V can also be used for this purpose. If a V is present in the mask, the decimal point is located at the V, and the position occupied by the V is deleted. If no V is present, the decimal point is located at the rightmost occurrence of the decimal point character (usually .). If neither a V nor a

decimal point character is present, the decimal point is assumed to be to the right of the rightmost character of the entire mask.

Although leading and trailing text in a mask is always transferred to the result field, text embedded between digit selectors is transferred only if the corresponding digits to the right and left are transferred.

For example, a value that is intended to represent a date can be formatted with an M field descriptor as follows:

Format	Data Value	Result
`M"99/99/99"`	`103179`	`|10/31/79|`

This is a comparison of the effects of using the 9 and Z as digit selectors. The minus sign in the preceding examples is the symbol that is automatically displayed for negative values in the absence of any specification to the contrary by a decoration. As shown in the preceding examples, a decimal point in the mask can be used for radix point alignment of the external field. Additional examples follow here:

Format	Data Value	Result
`3M<Z99.99>`	`-27.40, 12, 0`	`-27.40 12.00 00.00` `/\ /\ /\`
`3M<ZZ9.99>`	`-27.40, 12, 0`	`-27.40 12.00 0.00` `/\ /\ /\`
`3M<ZZZ.99>`	`-27.40, 12, 0`	`-27.40 12.00 00` `/\ /\ /\`

The “/\” marker is used to denote the boundaries of the output field.

In the example below, a comma specified as mask text is not displayed.

Format	Data Value	Result
`M'Z,ZZ9.99'`	`32.009`	`| 32.01|`

The “|”character is used to denote the boundaries of the output field.

Compare the different treatment of the embedded commas in these examples:

Data Values: `298738472, 389487.987, 666, 0.35`

Format One: `M<$ ZZZ,ZZZ,ZZ9 AND NO CENTS>`

Format Two: `M<$ 999,999,999 AND NO CENTS>`

Format One	Format Two
`$ 298,738,472 AND NO CENTS`	`$ 298,738,472 AND NO CENTS`

```
Format One                          Format Two
$      389,488 AND NO CENTS         $ 000,389,488 AND NO CENTS
$          666 AND NO CENTS         $ 000,000,666 AND NO CENTS
$            0 AND NO CENTS         $ 000,000,000 AND NO CENTS
```

The M edit descriptor can be useful in producing visually effective reports, by formatting values into patterns that are meaningful in terms of the data they represent. For example, assume that four arrays contain this data:

```
Amount    := 9758 21573 15532
Date      := 031777 091779 090579
District  := 'WEST','MIDWEST','SOUTH'
Telephone := 2135296800,2162296270,4047298400
```

This format can then be used to output the data as a table whose entries are in familiar forms. Assuming the elements are presented to the formatter in the order: the first elements of each array, followed by the second elements of each array, and so on, using this format:

```
M<$ZZ,ZZ9>,M<   Z9/99/99>,3X,A8,M<   (999) 999-9999>
```

the result would be:

```
$ 9,758     3/17/77    WEST       (213) 529-6800
$21,573     9/17/79    MIDWEST    (216) 229-6270
$15,532     9/05/79    SOUTH      (404) 729-8400
```

The O Edit Descriptor

The O edit descriptor is used to display or interpret data values in octal (base 8) integer form.

The O edit descriptor has these forms:

O*w* OR O*w.m*

- *w* an unsigned integer constant that defines the total field width and cannot exceed 255. The field processed is the *w* characters starting at the current position. After the field is processed, the current position is advanced by *w* characters.
- *m* an unsigned integer constant that defines the number of digits that must be present on output.

The O edit descriptor is used in the same manner as the I edit descriptor where the number base (*b*) is 8, except that the O edit descriptor always treats the internal data

value as unsigned. (See The I Edit Descriptor on page F-14.) For example, if the data item is an INT(16) in TAL, these conversions take place:

Format	Internal Value	Result
O6	10	| 12|
O6.6	18	|000022|
O6.4	-3	|177775|

The "|" character is used to denote the boundaries of the output field.

The Z Edit Descriptor

The Z edit descriptor is used to display or interpret data values in hexadecimal (base 16) integer form.

The Z edit descriptor has these forms:

Z*w* OR Z*w.m*

w an unsigned integer constant that defines the total field width and cannot exceed 255. The field processed is the *w* characters starting at the current position. After the field is processed, the current position is advanced by *w* characters.

m an unsigned integer constant that defines the number of digits that must be present on output.

The Z edit descriptor is used in the same manner as the I edit descriptor where the number base (*b*) is 16, except that the Z edit descriptor always treats the internal data value as unsigned. (See The I Edit Descriptor on page F-14) For example, if the data item is an INT(16) in TAL, these conversions take place:

Format	Internal Value	Result
Z6	20	| 14|
Z6.6	26	|00001A|
Z6.2	-3	| FFFD|

The "|" character is used to denote the boundaries of the output field.

Modifiers

Modifiers alter the normal effect of edit descriptors. Modifiers immediately precede the edit descriptor to which they apply. If modifiers immediately precede the left parenthesis of a group, the modifiers apply to each repeatable edit descriptor within the group. They are enclosed in brackets, and if more than one is present, they are separated by commas.

Note. Modifiers are effective only on output. If they are supplied for input, they have no effect.

Field-Blanking Modifiers (BN, BZ)

There are two modifiers for blanking fields:

BN blank field if null.

BZ blank field if equal to zero.

Although most edit descriptors cause a minimum number of characters to be output, a field-blanking modifier causes the entire field to be filled with spaces if the specified condition is met. The null value is the value addressed by the *nullptr* in the *variablelist* entry for the current data element.

Fill-Character Modifier (FL)

When an alphanumeric data element contains fewer characters than the field width specified by an Aw edit descriptor, when leading or trailing zero suppression is performed, or when embedded text in an M edit descriptor is not output because its neighboring digits are not output, a fill character is inserted in each appropriate character position in the output field. The fill character is normally a space, but the fill-character modifier can be used to specify any other character for this purpose. The fill-character modifier has the form:

FL *char*

char any single character, enclosed in quotation marks or apostrophes.

These are examples of fill-character replacement:

Format	Data Value	Result
`[FL'.']A10`	`'THEN'`	`|THEN......|`
`[RJ,FL">"]A7`	`'HERE'`	`|>>>HERE|`
`[FL"*"]M<$ZZ,ZZ9.99>`	`127.39`	`|$***127.39|`

The “|” character is used to denote the boundaries of the output field.

Overflow-Character Modifier (OC)

The overflow condition occurs if there are more characters to be placed into a field than there are positions provided by the edit descriptors. In the absence of any modifier or decoration to the contrary, if an overflow condition occurs in a numeric field, the field is filled with asterisks (*). This applies to the D, E, F, G, I, and M edit descriptors. The OC modifier can be used to substitute any other character for the asterisk as the overflow indicator character.

The OC modifier has the form:

OC *char*

char any single character, enclosed in quotation marks or apostrophes.

For example, the modifier [OC '!'] causes the output field to be filled with exclamation marks, instead of asterisks, if an overflow occurs:

Format	Data Value	Result
`[OC'!']I2`	`100`	`|!!|`

The "|" character is used to denote the boundaries of the output field.

Justification Modifiers (LJ, RJ)

The A edit descriptor normally displays the data left justified in its field.

The justification modifiers are:

LJ Left justify (normal)

RJ Right justify (data is displayed right justified)

The RJ and LJ modifiers are used with the A edit descriptor only.

Symbol-Substitution Modifier (SS)

The symbol-substitution modifier permits the user to replace certain standard symbols used by the formatter with other symbols. It can be used with the M edit descriptor to free the special characters 9, V, ., and Z for use as text characters in the mask. It can also be used with the D, E, F, and G edit descriptors to alter the standard characters they insert in the result field. The symbol substitution modifier has the form:

SS *symprs*

symprs one or more pairs of symbols enclosed in quotation marks or apostrophes. The first symbol in each pair is one of those in this table; the second is the symbol that is to replace it temporarily.

These formatting symbols can be altered by the SS modifier:

Symbol	Function
9	Digit selector (M format)
Z	Digit selector, zero suppression (M format)
V	Decimal alignment character (M format)
.	Decimal point (D, E, F, G, and M format)

These examples show how the SS modifier can be used to permit decimal values to be displayed as clock times, to follow European conventions (where a comma is used as the decimal point and periods are used as digit group separators), or to alter the function of the digit selectors in the M edit descriptor. When using the symbol substitution with a mask format, to obtain the function of one special character which is being altered by the symbol substitution, use the new character of the pair. With all other formats, use the old character of the pair; for example:

Data Value	Format	Result
`12.45`	`[SS".:"]F6.2`	`| 12:45|`
`12.45`	`[SS".:"]M<ZZZ:99>`	`| 12:45|`
`12345.67`	`[SS'.,']F10.2`	`|  12345,67|`
`103179`	`[SS<9X>]M<XX/XX/19XX>`	`|10/31/1979|`

The “|” character is used to denote the boundaries of the output field.

This table indicates which modifiers can be used with which edit descriptors (Y stands for yes, the combination is permitted).

	Edit Descriptors						
Modifiers	**A**	**E,D**	**F**	**G**	**I**	**L**	**M**
BZ,BN	Y	Y	Y	Y	Y	Y	Y
LJ,RJ	Y						
OC		Y	Y	Y	Y	Y	Y
FL	Y	Y	Y	Y	Y		Y
SS		Y	Y	Y			Y

Decorations

A decoration specifies a character string that can be added to the result field, the conditions under which the string is to be added, the location at which the string is to be added, and whether it is to be added before normal formatting is done or after it is completed.

You can use multiple decorations, separated by commas, with the same edit descriptor. Decorations are enclosed in brackets (together with any modifiers) and immediately precede the edit descriptor to which they apply. If modifiers immediately precede the left parenthesis of a group, the modifiers apply to each repeatable edit descriptor within the group.

When a field is processed, the floating decorations appear in the same order, left to right. If an edit descriptor within a group already has some decorations, the decorations that are applied to the group function as if they were placed to the right of the decorations already present. A decoration has the form:

{ M } { M } { N } ... { F } *string*

OR

{ N } { P } { P } { P } ... An *string* { Z } { Z } { O }

Character 1	Field condition specifier:	M	Minus
		N	Null
		O	Overflow
		P	Plus
		Z	Zero
Character 2	String location specifier:	A	Absolute
		F	Floating
		P	Prior

n an unsigned nonzero integer constant that specifies the actual character position within the field at which the string is to begin.

string any character string enclosed in quotation marks or apostrophes.

Note. Only location type An can be used in combination with the O condition.

Conditions

The condition specifier states that the string is to be added to the field if its value is minus, zero, positive, or null, or if a field overflow has occurred. A null condition takes precedence over negative, positive, and zero conditions; the overflow test is done after those for the other conditions, and therefore precedence is not significant. Alphanumeric data elements are considered to be positive or null only.

A decoration can have more than one condition specifier. If multiple condition specifiers are entered, an “or” condition is understood. For example, “ZPA2‘+’ ”

specifies that the string is to be inserted in the field if the data value is equal to or greater than zero.

Locations

The location specifier indicates where the string is to be added to the field.

The A specifier states that the string is to begin in absolute position n within the field. The leftmost position of the field is position 1.

The F specifier states that, once the number of data characters in the field has been established, the string is to occupy the position or positions (for right-justified fields) immediately to the left of the leftmost data character. This is reversed for left-justified elements.

The P specifier states that, before normal formatting, the string is to be inserted in the rightmost (for right-justified fields) end of the field; data characters are shifted to the left an appropriate number of positions. This is reversed for left-justified fields.

Processing

Decoration processing is as follows:

1. The data element is determined to have a negative, positive, zero, or null value; a null condition takes precedence over the other attributes.
2. If a P location decoration is specified and its condition is satisfied, its string is inserted in the field.
3. Normal formatting is performed.
4. If A or F decorations are specified and their conditions met, they are applied.
5. If an attempt is made to transfer more characters to the field than can be accommodated (in Step 2, 3, or 4), the overflow condition is set. If an overflow decoration has been specified, it is applied.

Note. Only location type An can be used with the O condition.

These examples illustrate these considerations:

Format	Data Value	Result
`[MF'<',MP'>',ZPP' ']F12.2`	`1000.00`	`|  1000.00 |`
`[MF'<',MP'>',ZPP' ']F12.2`	`-1000.00`	`| <1000.00>|`
`[MA1'CR',MPF'$']F12.2`	`1000.00`	`|  $1000.00|`
`[MA1'CR',MPF'$']F12.2`	`-100.00`	`|CR  $100.00|`
`[OA1'**OVERFLOW**']F12.2`	`1000000.00`	`|1000000.00|`
`[OA1'**OVERFLOW**']F12.2`	`10000000.00`	`|**OVERFLOW**|`

The “|” character is used to denote the boundaries of the output field.

Note. These decorations are automatically applied to any numeric edit descriptor (D, E, F, G, I, or M) for which no decoration has been specified:

MF'-'

OA1'*** ... *' (The number of asterisks is equal to the number of characteristics in the field width.)

However, if any decoration with a condition code relating to the sign of the data is specified, the automatic MF'-' decoration no longer applies; if negative-value indication is desired, you must supply the appropriate decoration. If any decoration with a condition code relating to overflow is specified, the automatic OA1'***...*' decoration no longer applies. If MF'' is specified (that is, with no text string), then the default MF'-' is applied.

As an example of how decorations apply to a group of edit descriptors, these formats give the same results:

Format

```
[MF'-'](F10.2,[MZF'**']F10.2)
[MF'-']F10.2,[MZF'**',MF'-']F10.2
```

Using the format above:

Data Values	Results		
0,0	0.00	**0.00	
	/\	/\	/\
1,1	1.00	1.00	
	/\	/\	/\
-1,-1	-1.00	**-1.00	
	/\	/\	/\

The “/\” marker is used to denote the boundaries of the output field.

List-Directed Formatting

List-directed formatting provides the data conversion capabilities of the formatter without requiring the specification of a format. The FORMATDATA procedure determines the details of the data conversion, based on the types of the data elements. This is particularly convenient for input because the list-directed formatting rules provide for free-format input of data values rather than requiring data to be supplied in fixed fields. There are fewer advantages to using list-directed formatting for output because the output data is not necessarily arranged in a convenient readable form.

The characters in one or more list-directed buffers constitute a sequence of data-list items and value separators. Each value is either a constant, a null value, or one of these forms:

*r***c* *r**

r is an unsigned, nonzero, integer constant.

*r***c* form is equivalent to *r* successive appearances of the constant *c*.

*r** form is equivalent to *r* successive null values.

Neither of these forms can contain embedded blanks, except where permitted with the constant *c*.

List-Directed Input

All input forms that are acceptable to FORMATDATA when directed by a format are acceptable for list-directed input, with these exceptions:

- When the data element is a complex variable, the input form consists of a left parenthesis followed by an ordered pair of numeric input fields separated by a comma and followed by a right parenthesis.
- When the data element is a logical variable, the input form must not include either slashes or commas among the optional characters for the L editing.
- When the data element is a character variable, the input form consists of a string of characters enclosed in apostrophes. The blank, comma, and slash may appear in the string of characters.
- A null value is specified by having no characters other than blanks between successive value separators, no characters preceding the first value separator in the first buffer, or the *r** form. A null value has no effect on the value of the corresponding data element. The input list item retains its previous value. A single null value must represent an entire complex constant (not just part of it).

If a slash value separator is encountered during the processing of a buffer, data conversion is terminated. If there are additional elements in the data list, the effect is as if null values had been supplied for them.

On input, a value separator is one of these:

- A comma or slash optionally preceded or optionally followed by one or more contiguous blanks (except within a character constant).
- One or more contiguous blanks between two constants or following the last constant (except embedded blanks surrounding the real or imaginary part of a complex constant).
- The end of the buffer (except within a character constant).

List-Directed Output

Output forms that are produced by list-directed output are the same as that required for input with these exceptions:

- The end of a buffer can occur between the comma and the imaginary part of a complex constant only if the entire constant is as long as, or longer than, an entire buffer. The only embedded blanks permitted within a complex constant are between the comma and the end of a buffer, and one blank at the beginning of the next buffer.
- Character values are displayed without apostrophes.
- If two or more successive character values in an output record produced have identical values, the FORMATDATA procedure produces a repeated constant of the form *r***c* instead of the sequence of identical values.
- Slashes, as value separators, and null values are not produced by list-directed output.

For output, the value separator is a single blank. A value separator is not produced between or adjacent to character values.

G

Superseded Guardian Procedure Calls and Their Replacements

This appendix contains these tables listing superseded Guardian procedures and their replacements:

- Superseded Guardian Procedures and Their Replacements (H06.03)
- Table G-2, Superseded Guardian Procedures and Their Replacements (G00)
- Table G-3, Superseded Guardian Procedures and Their Replacements (D40)
- Table G-4, Superseded Guardian Procedures and Their Replacements (D30)
- Table G-5, Superseded C-Series Guardian Procedures and Their Replacements (D-Series)

Table G-1 lists the Guardian procedures that are superseded beginning in the H06.03 version of the operating system and indicates the procedures that replace them

.

Table G-1. Superseded Guardian Procedures and Their Replacements (H06.03)

Superseded Procedure	Replacement Procedure
DELAY	PROCESS_DELAY_

Table G-2 lists the Guardian procedures that are superseded beginning in the G00 version of the operating system and indicates the procedures that replace them.

Table G-2. Superseded Guardian Procedures and Their Replacements (G00)

Superseded Procedure	Replacement Procedure
DEVICE_GETINFOBYLDEV_	CONFIG_GETINFO_BYLDEV_
DEVICE_GETINFOBYNAME_	CONFIG_GETINFO_BYNAME_
DISK_REFRESH_	(not needed)
REFRESH	(not needed)

Table G-3 lists the Guardian procedures that are superseded beginning in the D40 version of the operating system and indicates the procedures that replace them. The superseded procedures continue to be supported for TNS processes. With the exception of the PROCESS_CREATE_ procedure, these superseded procedures cannot be called by TNS/R native processes.

Table G-3. Superseded Guardian Procedures and Their Replacements (D40)

This procedure	Is not defined for TNS/R native processes; use the procedure
ARMTRAP	SIGACTION_INIT_
CHECKPOINT	CHECKPOINTX
CHECKPOINTMANY	CHECKPOINTMANYX
CURRENTSPACE	(No replacement is needed)
FORMATDATA	FORMATDATAX
LASTADDR	ADDRESS_DELIMIT_
LASTADDRX	ADDRESS_DELIMIT_
PROCESS_CREATE_*	PROCESS_LAUNCH_
XBNDSTEST	REFPARAM_BOUNDSCHECK_
XSTACKTEST	HEADROOM_ENSURE_

*This procedure can be called by TNS/R native processes.

Table G-4 lists the Guardian procedures that are superseded beginning in the D30 version of the operating system and indicates the new procedures that replace them. The superseded procedures continue to be supported for compatibility.

Table G-4. Superseded Guardian Procedures and Their Replacements (D30)

Superseded Procedure	Replacement Procedure
DEFINEPOOL	*POOL procedures are replaced by POOL_* procedures. There is no one-for-one replacement.
GETPOOL	*POOL procedures are replaced by POOL_* procedures. There is no one-for-one replacement.
GROUPIDTOGROUPNAME	GROUP_GETINFO_
GROUPNAMETOGROUPID	GROUP_GETINFO_
PUTPOOL	*POOL procedures are replaced by POOL_* procedures. There is no one-for-one replacement.
RESIZEPOOL	*POOL procedures are replaced by POOL_* procedures. There is no one-for-one replacement.
USERDEFAULTS	USER_GETINFO_
USERIDTOUSERNAME	USER_GETINFO_
USERNAMETOUSERID	USER_GETINFO_
VERIFYUSER	USER_AUTHENTICATE_ and USER_GETINFO_

Table G-5 lists the C-series Guardian procedures that are superseded beginning in the D-series RVU of the operating system and indicates the new procedures that replace them. The superseded procedures continue to be supported for compatibility.

Table G-5. Superseded C-Series Guardian Procedures and Their Replacements (D-Series) (page 1 of 3)

Superseded Procedure	Replacement Procedure
ABEND	PROCESS_STOP_
ACTIVATEPROCESS	PROCESS_ACTIVATE_
ALLOCATESEGMENT	SEGMENT_ALLOCATE_
ALTER	FILE_ALTERLIST_
ALTERPRIORITY	PROCESS_SETINFO_
CHECKALLOCATESEGMENT	SEGMENT_ALLOCATE_CHKPT_
CHECKCLOSE	FILE_CLOSE_CHKPT_
CHECKDEALLOCATESEGMENT	SEGMENT_DEALLOCATE_CHKPT_
CHECKOPEN	FILE_OPEN_CHKPT_
CLOSE	FILE_CLOSE_
CLOSEEDIT	CLOSEEDIT_
CONVERTPROCESSNAME	FILENAME_RESOLVE_
CREATE	FILE_CREATE[LIST]_
CREATEPROCESSNAME	PROCESSNAME_CREATE_
CREATEREMOTENAME	PROCESSNAME_CREATE_
CREATORACCESSID	PROCESS_GETINFO[LIST]_
DEALLOCATESEGMENT	SEGMENT_DEALLOCATE_
DEBUGPROCESS	PROCESS_DEBUG_
DEVICEINFO[2]	FILE_GETINFOLISTBYNAME_ FILE_GETINFOBYNAME_
DISKINFO	FILE_GETINFOLISTBYNAME_
FILEINFO	FILE_GETINFO[LIST][BYNAME]_
FILEINQUIRE	FILE_GETINFO[LIST][BYNAME]_
FILERECINFO	FILE_GETINFO[LIST][BYNAME]_
FNAME32COLLAPSE	(not needed)
FNAME32EXPAND	FILENAME_SCAN_ and FILENAME_RESOLVE_
FNAME32TOFNAME	(not needed)
FNAMECOLLAPSE	(not needed)

Table G-5. Superseded C-Series Guardian Procedures and Their Replacements (D-Series) (page 2 of 3)

Superseded Procedure	Replacement Procedure
FNAMECOMPARE	FILENAME_COMPARE_
FNAMEEXPAND	FILENAME_SCAN_ and FILENAME_RESOLVE_
FNAMETOFNAME32	(not needed)
GETCRTPID	PROCESS_GETINFO[LIST]_
GETDEVNAME	DEVICE_GETINFOBYLDEV_ or FILENAME_FINDNEXT_
GETPPDENTRY	PROCESS_GETPAIRINFO_
GETREMOTECRTPID	PROCESS_GETINFO[LIST]_
GETSYSTEMNAME	NODENUMBER_TO_NODENAME_
LASTADDR[X]	ADDRESS_DELIMIT_
LASTRECEIVE	FILE_GETRECEIVEINFO_
LOCATESYSTEM	NODENAME_TO_NODENUMBER_
LOCKINFO	FILE_GETLOCKINFO_
LOOKUPPROCESSNAME	PROCESS_GETPAIRINFO_
MOM	PROCESS_GETINFO[LIST]_
MYGMOM	PROCESS_GETINFO[LIST]_
MYPID	PROCESSHANDLE_GETMINE_ and PROCESSHANDLE_DECOMPOSE_
MYSYSTEMNUMBER	NODENAME_TO_NODENUMBER_ or PROCESSHANDLE_GETMINE_ and PROCESSHANDLE_DECOMPOSE_
MYTERM	PROCESS_GETINFO[LIST]_
NEWPROCESS	PROCESS_CREATE_ and PROCESS_LAUNCH_
NEWPROCESSNOWAIT	PROCESS_CREATE_ and PROCESS_LAUNCH_
NEXTFILENAME	FILENAME_FINDNEXT_
OPEN	FILE_OPEN_
OPENEDIT	OPENEDIT_
OPENINFO	FILE_GETOPENINFO_
PRIORITY	PROCESS_SETINFO_ or PROCESS_GETINFO[LIST]_
PROCESSACCESSID	PROCESS_GETINFO[LIST]_

Table G-5. Superseded C-Series Guardian Procedures and Their Replacements (D-Series) (page 3 of 3)

Superseded Procedure	Replacement Procedure
PROCESSFILESECURITY	PROCESS_SETINFO_ or PROCESS_GETINFOLIST_
PROCESSINFO	PROCESS_GETINFO[LIST]_
PROCESSTIME	PROCESS_GETINFO[LIST]_
PROGRAMFILENAME	PROCESS_GETINFO[LIST]_
PURGE	FILE_PURGE_
RECEIVEINFO	FILE_GETRECEIVEINFO_
REFRESH	DISK_REFRESH_
RENAME	FILE_RENAME_
SEGMENTSIZE	SEGMENT_GET[BACKUP]INFO_
SENDBREAKMESSAGE	BREAKMESSAGE_SEND_
SETMYTERM	PROCESS_SETSTRINGINFO_
SHIFTSTRING	STRING_UPSHIFT_
STEPMOM	PROCESS_SETINFO_
STOP	PROCESS_STOP_
SUSPENDPROCESS	PROCESS_SUSPEND_
USESEGMENT	SEGMENT_USE_

Documented Guardian Procedures

This list shows all documented Guardian procedures and the manuals in which they are described:

ABEND	*Guardian Procedure Calls Reference Manual*
ABORTTRANSACTION	*TMF Application Programmer's Guide*
ACTIVATEPROCESS	*Guardian Procedure Calls Reference Manual*
ACTIVATERECEIVETRANSID	*TMF Application Programmer's Guide*
ADDDSTTRANSITION	*Guardian Procedure Calls Reference Manual*
ADDRESS_DELIMIT_	*Guardian Procedure Calls Reference Manual*
ADDRTOPROCNAME	*Guardian Procedure Calls Reference Manual*
ALLOCATESEGMENT	*Guardian Procedure Calls Reference Manual*
ALTER	*Guardian Procedure Calls Reference Manual*
ALTERPRIORITY	*Guardian Procedure Calls Reference Manual*
APS_*	*OSI/AS Programming Guide*
ARMTRAP	*Guardian Procedure Calls Reference Manual*
AWAITIO	*Guardian Procedure Calls Reference Manual*
AWAITIOX	*Guardian Procedure Calls Reference Manual*
BACKSPACEEDIT	*Guardian Procedure Calls Reference Manual*
BEGINTRANSACTION	*TMF Application Programmer's Guide*
BINSEM_CLOSE_	*Guardian Procedure Calls Reference Manual*
BINSEM_CREATE_	*Guardian Procedure Calls Reference Manual*
BINSEM_FORCELOCK_	*Guardian Procedure Calls Reference Manual*
BINSEM_LOCK_	*Guardian Procedure Calls Reference Manual*
BINSEM_OPEN_	*Guardian Procedure Calls Reference Manual*
BINSEM_UNLOCK_	*Guardian Procedure Calls Reference Manual*
BREAKMESSAGE_SEND_	*Guardian Procedure Calls Reference Manual*
CANCEL	*Guardian Procedure Calls Reference Manual*
CANCELPROCESSTIMEOUT	*Guardian Procedure Calls Reference Manual*
CANCELREQ	*Guardian Procedure Calls Reference Manual*

CANCELTIMEOUT	*Guardian Procedure Calls Reference Manual*
CHANGELIST	*Guardian Procedure Calls Reference Manual*
CHECK^BREAK	*Guardian Procedure Calls Reference Manual*
CHECK^FILE	*Guardian Procedure Calls Reference Manual*
CHECKALLOCATESEGMENT	*Guardian Procedure Calls Reference Manual*
CHECKCLOSE	*Guardian Procedure Calls Reference Manual*
CHECKDEALLOCATESEGMENT	*Guardian Procedure Calls Reference Manual*
CHECKDEFINE	*Guardian Procedure Calls Reference Manual*
CHECKMONITOR	*Guardian Procedure Calls Reference Manual*
CHECKOPEN	*Guardian Procedure Calls Reference Manual*
CHECKPOINT	*Guardian Procedure Calls Reference Manual*
CHECKPOINTMANY	*Guardian Procedure Calls Reference Manual*
CHECKPOINTMANYX	*Guardian Procedure Calls Reference Manual*
CHECKPOINTX	*Guardian Procedure Calls Reference Manual*
CHECKRESIZESEGMENT	*Guardian Procedure Calls Reference Manual*
CHECKSETMODE	*Guardian Procedure Calls Reference Manual*
CHECKSWITCH	*Guardian Procedure Calls Reference Manual*
CHILD_LOST_	*Guardian Procedure Calls Reference Manual*
CLOSE	*Guardian Procedure Calls Reference Manual*
CLOSE^FILE	*Guardian Procedure Calls Reference Manual*
CLOSEALLEDIT	*Guardian Procedure Calls Reference Manual*
CLOSEEDIT	*Guardian Procedure Calls Reference Manual*
CLOSEEDIT_	*Guardian Procedure Calls Reference Manual*
COMPLETEIOEDIT	*Guardian Procedure Calls Reference Manual*
COMPRESSEDIT	*Guardian Procedure Calls Reference Manual*
COMPUTEJULIANDAYNO	*Guardian Procedure Calls Reference Manual*
COMPUTETIMESTAMP	*Guardian Procedure Calls Reference Manual*
COMPUTETRANSID	*TMF Application Programmer's Guide*
CONFIG_GETINFO_BYLDEV_	*Guardian Procedure Calls Reference Manual*
CONFIG_GETINFO_BYLDEV2_	*Guardian Procedure Calls Reference Manual*

CONFIG_GETINFO_BYNAME_	*Guardian Procedure Calls Reference Manual*
CONFIG_GETINFO_BYNAME2_	*Guardian Procedure Calls Reference Manual*
CONTIME	*Guardian Procedure Calls Reference Manual*
CONTROL	*Guardian Procedure Calls Reference Manual*
CONTROLBUF	*Guardian Procedure Calls Reference Manual*
CONTROLMESSAGESYSTEM	*Guardian Procedure Calls Reference Manual*
CONVERTASCIIEBCDIC	*Guardian Procedure Calls Reference Manual*
CONVERTPROCESSNAME	*Guardian Procedure Calls Reference Manual*
CONVERTPROCESSTIME	*Guardian Procedure Calls Reference Manual*
CONVERTTIMESTAMP	*Guardian Procedure Calls Reference Manual*
CPRL_*	*SQL/MP Programming Manual for C, SQL/MP Programming Manual for COBOL85*
CPU_GETINFOLIST_ (Alternative name for PROCESSOR_GETINFOLIST_)	*Guardian Procedure Calls Reference Manual*
CPUTIMES	*Guardian Procedure Calls Reference Manual*
CREATE	*Guardian Procedure Calls Reference Manual*
CREATEPROCESSNAME	*Guardian Procedure Calls Reference Manual*
CREATEREMOTENAME	*Guardian Procedure Calls Reference Manual*
CREATORACCESSID	*Guardian Procedure Calls Reference Manual*
CRTPID_TO_PROCESSHANDLE_	*Guardian Procedure Calls Reference Manual*
CURRENTSPACE	*Guardian Procedure Calls Reference Manual*
DAYOFWEEK	*Guardian Procedure Calls Reference Manual*
DEALLOCATESEGMENT	*Guardian Procedure Calls Reference Manual*
DEBUG	*Guardian Procedure Calls Reference Manual*
DEBUGPROCESS	*Guardian Procedure Calls Reference Manual*
DEFINEADD	*Guardian Procedure Calls Reference Manual*
DEFINEDELETE	*Guardian Procedure Calls Reference Manual*
DEFINEDELETEALL	*Guardian Procedure Calls Reference Manual*
DEFINEINFO	*Guardian Procedure Calls Reference Manual*
DEFINELIST	*Guardian Procedure Calls Reference Manuall*

DEFINEMODE	*Guardian Procedure Calls Reference Manual*
DEFINENEXTNAME	*Guardian Procedure Calls Reference Manual*
DEFINEPOOL	*Guardian Procedure Calls Reference Manual*
DEFINEREADATTR	*Guardian Procedure Calls Reference Manual*
DEFINERESTORE	*Guardian Procedure Calls Reference Manual*
DEFINERESTOREWORK	*Guardian Procedure Calls Reference Manual*
DEFINERESTOREWORK2	*Guardian Procedure Calls Reference Manual*
DEFINESAVE	*Guardian Procedure Calls Reference Manual*
DEFINESAVEWORK	*Guardian Procedure Calls Reference Manual*
DEFINESAVEWORK2	*Guardian Procedure Calls Reference Manual*
DEFINESETATTR	*Guardian Procedure Calls Reference Manual*
DEFINESETLIKE	*Guardian Procedure Calls Reference Manual*
DEFINEVALIDATEWORK	*Guardian Procedure Calls Reference Manual*
DELAY	*Guardian Procedure Calls Reference Manual*
DELETEEDIT	*Guardian Procedure Calls Reference Manual*
DEVICE_GETINFOBYLDEV_	*Guardian Procedure Calls Reference Manual*
DEVICE_GETINFOBYNAME_	*Guardian Procedure Calls Reference Manual*
DEVICEINFO	*Guardian Procedure Calls Reference Manual*
DEVICEINFO2	*Guardian Procedure Calls Reference Manual*
DISK_REFRESH_	*Guardian Procedure Calls Reference Manual*
DISKINFO	*Guardian Procedure Calls Reference Manual*
DNUMIN	*Guardian Procedure Calls Reference Manual*
DNUMOUT	*Guardian Procedure Calls Reference Manual*
DST_TRANSITION_ADD_	*Guardian Procedure Calls Reference Manual*
DST_TRANSITION_DELETE_	*Guardian Procedure Calls Reference Manual*
DST_TRANSITION_MODIFY_	*Guardian Procedure Calls Reference Manual*
DST_GETINFO_	*Guardian Procedure Calls Reference Manual*
EDITREAD	*Guardian Procedure Calls Reference Manual*
EDITREADINIT	*Guardian Procedure Calls Reference Manual*
EMSADDSUBJECT	*EMS Manual*

EMSADDSUBJECTMAP	*EMS Manual*
EMSADDTOKENMAPS	*EMS Manual*
EMSADDTOKENS	*EMS Manual*
EMSGET	*EMS Manual*
EMSGETTKN	*EMS Manual*
EMSINIT	*EMS Manual*
EMSINITMAP	*EMS Manual*
EMSTEXT	*EMS Manual*
ENDTRANSACTION	*TMF Application Programmer's Guide*
ENFORMFINISH	*Enform User's Guide*
ENFORMRECEIVE	*Enform User's Guide*
ENFORMSTART	*Enform User's Guide*
ERRNO_GET	*Guardian Procedure Calls Reference Manual*
EXTENDEDIT	*Guardian Procedure Calls Reference Manual*
FILE_ALTERLIST_	*Guardian Procedure Calls Reference Manual*
FILE_CLOSE_	*Guardian Procedure Calls Reference Manual*
FILE_CLOSE_CHKPT_	*Guardian Procedure Calls Reference Manual*
FILE_CREATE_	*Guardian Procedure Calls Reference Manual*
FILE_CREATELIST_	*Guardian Procedure Calls Reference Manual*
FILE_GETINFO_	*Guardian Procedure Calls Reference Manual*
FILE_GETINFOBYNAME_	*Guardian Procedure Calls Reference Manual*
FILE_GETINFOLIST_	*Guardian Procedure Calls Reference Manual*
FILE_GETINFOLISTBYNAME_	*Guardian Procedure Calls Reference Manual*
FILE_GETLOCKINFO_	*Guardian Procedure Calls Reference Manual*
FILE_GETOPENINFO_	*Guardian Procedure Calls Reference Manual*
FILE_GETRECEIVEINFO_	*Guardian Procedure Calls Reference Manual*
FILE_OPEN_	*Guardian Procedure Calls Reference Manual*
FILE_OPEN_CHKPT_	*Guardian Procedure Calls Reference Manual*
FILE_PURGE_	*Guardian Procedure Calls Reference Manual*
FILE_RENAME_	*Guardian Procedure Calls Reference Manual*

FILEERROR	*Guardian Procedure Calls Reference Manual*
FILEINFO	*Guardian Procedure Calls Reference Manual*
FILEINQUIRE	*Guardian Procedure Calls Reference Manual*
FILENAME_COMPARE_	*Guardian Procedure Calls Reference Manual*
FILENAME_DECOMPOSE_	*Guardian Procedure Calls Reference Manual*
FILENAME_EDIT_	*Guardian Procedure Calls Reference Manual*
FILENAME_FINDFINISH_	*Guardian Procedure Calls Reference Manual*
FILENAME_FINDNEXT_	*Guardian Procedure Calls Reference Manual*
FILENAME_FINDSTART_	*Guardian Procedure Calls Reference Manual*
FILENAME_MATCH_	*Guardian Procedure Calls Reference Manual*
FILENAME_RESOLVE_	*Guardian Procedure Calls Reference Manual*
FILENAME_SCAN_	*Guardian Procedure Calls Reference Manual*
FILENAME_TO_OLDFILENAME_	*Guardian Procedure Calls Reference Manual*
FILENAME_TO_PATHNAME_	*Guardian Procedure Calls Reference Manual*
FILENAME_TO_PROCESSHANDLE_	*Guardian Procedure Calls Reference Manual*
FILENAME_UNRESOLVE_	*Guardian Procedure Calls Reference Manual*
FILERECINFO	*Guardian Procedure Calls Reference Manual*
FIXSTRING	*Guardian Procedure Calls Reference Manual*
FNAME32COLLAPSE	*Guardian Procedure Calls Reference Manual*
FNAME32EXPAND	*Guardian Procedure Calls Reference Manual*
FNAME32TOFNAME	*Guardian Procedure Calls Reference Manual*
FNAMECOLLAPSE	*Guardian Procedure Calls Reference Manual*
FNAMECOMPARE	*Guardian Procedure Calls Reference Manual*
FNAMEEXPAND	*Guardian Procedure Calls Reference Manual*
FNAMETOFNAME32	*Guardian Procedure Calls Reference Manual*
FORMATCONVERT	*Guardian Procedure Calls Reference Manual*
FORMATCONVERTX	*Guardian Procedure Calls Reference Manual*
FORMATDATA	*Guardian Procedure Calls Reference Manual*
FORMATDATAX	*Guardian Procedure Calls Reference Manual*
FTM_*	*OSI/FTAM Programming Reference Manual*

GETCPCBINFO	*Guardian Procedure Calls Reference Manual*
GETCRTPID	*Guardian Procedure Calls Reference Manual*
GETDEVNAME	*Guardian Procedure Calls Reference Manual*
GETINCREMENTEDIT	*Guardian Procedure Calls Reference Manual*
GETPOOL	*Guardian Procedure Calls Reference Manual*
GETPOOL_PAGE_	*Guardian Procedure Calls Reference Manual*
GETPOSITIONEDIT	*Guardian Procedure Calls Reference Manual*
GETPPDENTRY	*Guardian Procedure Calls Reference Manual*
GETREMOTECRTPID	*Guardian Procedure Calls Reference Manual*
GETSYNCINFO	*Guardian Procedure Calls Reference Manual*
GETSYSTEMNAME	*Guardian Procedure Calls Reference Manual*
GETTMPNAME	*TMF Application Programmer's Guide*
GETTRANSID	*TMF Application Programmer's Guide*
GIVE^BREAK	*Guardian Procedure Calls Reference Manual*
GROUP_GETINFO_	*Guardian Procedure Calls Reference Manual*
GROUPIDTOGROUPNAME	*Guardian Procedure Calls Reference Manual*
GROUPMEMBER_GETNEXT_	*Guardian Procedure Calls Reference Manual*
GROUPNAMETOGROUPID	*Guardian Procedure Calls Reference Manual*
HALTPOLL	*Guardian Procedure Calls Reference Manual*
HEADROOM_ENSURE_	*Guardian Procedure Calls Reference Manual*
HEAPSORT	*Guardian Procedure Calls Reference Manual*
HEAPSORTX_	*Guardian Procedure Calls Reference Manual*
HIST_FORMAT_	*Guardian Procedure Calls Reference Manual*
HIST_GETPRIOR_	*Guardian Procedure Calls Reference Manual*
HIST_INIT_	*Guardian Procedure Calls Reference Manual*
INCREMENTEDIT	*Guardian Procedure Calls Reference Manual*
INITIALIZEEDIT	*Guardian Procedure Calls Reference Manual*
INITIALIZER	*Guardian Procedure Calls Reference Manual*
INTERPRETINTERVAL	*Guardian Procedure Calls Reference Manual*
INTERPRETJULIANDAYNO	*Guardian Procedure Calls Reference Manual*

INTERPRETTIMESTAMP	*Guardian Procedure Calls Reference Manual*
INTERPRETTRANSID	*TMF Application Programmer's Guide*
JULIANTIMESTAMP	*Guardian Procedure Calls Reference Manual*
KEYPOSITION	*Guardian Procedure Calls Reference Manual*
KEYPOSITIONX	*Guardian Procedure Calls Reference Manual*
LABELEDTAPESUPPORT	*Guardian Procedure Calls Reference Manual*
LASTADDR	*Guardian Procedure Calls Reference Manual*
LASTADDRX	*Guardian Procedure Calls Reference Manual*
LASTRECEIVE	*Guardian Procedure Calls Reference Manual*
LOCATESYSTEM	*Guardian Procedure Calls Reference Manual*
LOCKFILE	*Guardian Procedure Calls Reference Manual*
LOCKINFO	*Guardian Procedure Calls Reference Manual*
LOCKREC	*Guardian Procedure Calls Reference Manual*
LONGJMP	*Guardian Procedure Calls Reference Manual*
LOOKUPPROCESSNAME	*Guardian Procedure Calls Reference Manual*
MBCS_ANY_KATAKANA_	*Guardian Procedure Calls Reference Manual*
MBCS_CHAR_	*Guardian Procedure Calls Reference Manual*
MBCS_CHARSIZE_	*Guardian Procedure Calls Reference Manual*
MBCS_CHARSTRING_	*Guardian Procedure Calls Reference Manual*
MBCS_CODESETS_SUPPORTED_	*Guardian Procedure Calls Reference Manual*
MBCS_DEFAULTCHARSET_	*Guardian Procedure Calls Reference Manual*
MBCS_EXTERNAL_TO_COMPAQ_	*Guardian Procedure Calls Reference Manual*
MBCS_FORMAT_CRT_FIELD_	*Guardian Procedure Calls Reference Manual*
MBCS_FORMAT_ITI_BUFFER_	*Guardian Procedure Calls Reference Manual*
MBCS_MB_TO_SB_	*Guardian Procedure Calls Reference Manual*
MBCS_REPLACEBLANK_	*Guardian Procedure Calls Reference Manual*
MBCS_SB_TO_MB_	*Guardian Procedure Calls Reference Manual*
MBCS_SHIFTSTRING_	*Guardian Procedure Calls Reference Manual*
MBCS_COMPAQ_TO_EXTERNAL_	*Guardian Procedure Calls Reference Manual*
MBCS_TESTBYTE_	*Guardian Procedure Calls Reference Manual*

MBCS_TRIMFRAGMENT_	*Guardian Procedure Calls Reference Manual*
MEASCLOSE	*Measure Reference Manual*
MEASCONFIGURE	*Measure Reference Manual*
MEASCONTROL	*Measure Reference Manual*
MEASCOUNTERBUMP	*Measure Reference Manual*
MEASCOUNTERBUMPINIT	*Measure Reference Manual*
MEASGETVERSION	*Measure Reference Manual*
MEASINFO	*Measure Reference Manual*
MEASMONCONTROL	*Measure Reference Manual*
MEASMONSTATUS	*Measure Reference Manual*
MEASOPEN	*Measure Reference Manual*
MEASREAD	*Measure Reference Manual*
MEASREAD_DIFF_	*Measure Reference Manual*
MEASREADACTIVE	*Measure Reference Manual*
MEASREADCONF	*Measure Reference Manual*
MEASSTATUS	*Measure Reference Manual*
MEASWRITE_DIFF_	*Measure Reference Manual*
MESSAGESTATUS	*Guardian Procedure Calls Reference Manual*
MESSAGESYSTEMINFO	*Guardian Procedure Calls Reference Manual*
MFM_AWAITIOX_	*OSI/AS Programming Guide*
MFM_CANCELREQ_	*OSI/AS Programming Guide*
MOM	*Guardian Procedure Calls Reference Manual*
MONITORCPUS	*Guardian Procedure Calls Reference Manual*
MONITORNET	*Guardian Procedure Calls Reference Manual*
MONITORNEW	*Guardian Procedure Calls Reference Manual*
MOVEX	*Guardian Procedure Calls Reference Manual*
MYGMOM	*Guardian Procedure Calls Reference Manual*
MYPID	*Guardian Procedure Calls Reference Manual*
MYPROCESSTIME	*Guardian Procedure Calls Reference Manual*
MYSYSTEMNUMBER	*Guardian Procedure Calls Reference Manual*

MYTERM	*Guardian Procedure Calls Reference Manual*
NEWPROCESS	*Guardian Procedure Calls Reference Manual*
NEWPROCESSNOWAIT	*Guardian Procedure Calls Reference Manual*
NEXTFILENAME	*Guardian Procedure Calls Reference Manual*
NO^ERROR	*Guardian Procedure Calls Reference Manual*
NODE_GETCOLDLOADINFO_	*Guardian Procedure Calls Reference Manual*
NODENAME_TO_NODENUMBER_	*Guardian Procedure Calls Reference Manual*
NODENUMBER_TO_NODENAME_	*Guardian Procedure Calls Reference Manual*
NUMBEREDIT	*Guardian Procedure Calls Reference Manual*
NUMIN	*Guardian Procedure Calls Reference Manual*
NUMOUT	*Guardian Procedure Calls Reference Manual*
OBJFILE_GETINFOLIST_	*Guardian Procedure Calls Reference Manual*
OLDFILENAME_TO_FILENAME_	*Guardian Procedure Calls Reference Manual*
OLDSYSMSG_TO_NEWSYSMSG_	*Guardian Procedure Calls Reference Manual*
OPEN	*Guardian Procedure Calls Reference Manual*
OPEN^FILE	*Guardian Procedure Calls Reference Manual*
OPENEDIT	*Guardian Procedure Calls Reference Manual*
OPENEDIT_	*Guardian Procedure Calls Reference Manual*
OPENER_LOST_	*Guardian Procedure Calls Reference Manual*
OPENINFO	*Guardian Procedure Calls Reference Manual*
OSS_PID_NULL_	*Guardian Procedure Calls Reference Manual*
PACKEDIT	*Guardian Procedure Calls Reference Manual*
PATHNAME_TO_FILENAME_	*Guardian Procedure Calls Reference Manual*
POOL_CHECK_	*Guardian Procedure Calls Reference Manual*
POOL_DEFINE_	*Guardian Procedure Calls Reference Manual*
POOL_GETINFO_	*Guardian Procedure Calls Reference Manual*
POOL_GETSPACE_	*Guardian Procedure Calls Reference Manual*
POOL_GETSPACE_PAGE_	*Guardian Procedure Calls Reference Manual*
POOL_PUTSPACE_	*Guardian Procedure Calls Reference Manual*
POOL_RESIZE_	*Guardian Procedure Calls Reference Manual*

POSITION	*Guardian Procedure Calls Reference Manual*
POSITIONEDIT	*Guardian Procedure Calls Reference Manual*
PRINTCOMPLETE	*Spooler Programmer's Guide*
PRINTCOMPLETE2	*Spooler Programmer's Guide*
PRINTINFO	*Spooler Programmer's Guide*
PRINTINIT	*Spooler Programmer's Guide*
PRINTINIT2	*Spooler Programmer's Guide*
PRINTREAD	*Spooler Programmer's Guide*
PRINTREADCOMMAND	*Spooler Programmer's Guide*
PRINTSTART	*Spooler Programmer's Guide*
PRINTSTART2	*Spooler Programmer's Guide*
PRINTSTATUS	*Spooler Programmer's Guide*
PRINTSTATUS2	*Spooler Programmer's Guide*
PRIORITY	*Guardian Procedure Calls Reference Manual*
PROCESS_ACTIVATE_	*Guardian Procedure Calls Reference Manual*
PROCESS_CREATE_	*Guardian Procedure Calls Reference Manual*
PROCESS_DELAY_	*Guardian Procedure Calls Reference Manual*
PROCESS_DEBUG_	*Guardian Procedure Calls Reference Manual*
PROCESS_GETINFO_	*Guardian Procedure Calls Reference Manual*
PROCESS_GETINFOLIST_	*Guardian Procedure Calls Reference Manual*
PROCESS_GETPAIRINFO_	*Guardian Procedure Calls Reference Manual*
PROCESS_LAUNCH_	*Guardian Procedure Calls Reference Manual*
PROCESS_SETINFO_	*Guardian Procedure Calls Reference Manual*
PROCESS_SETSTRINGINFO_	*Guardian Procedure Calls Reference Manual*
PROCESS_SPAWN_	*Guardian Procedure Calls Reference Manual*
PROCESS_STOP_	*Guardian Procedure Calls Reference Manual*
PROCESS_SUSPEND_	*Guardian Procedure Calls Reference Manual*
PROCESSACCESSID	*Guardian Procedure Calls Reference Manual*
PROCESSFILESECURITY	*Guardian Procedure Calls Reference Manual*
PROCESSHANDLE_COMPARE_	*Guardian Procedure Calls Reference Manual*

PROCESSHANDLE_DECOMPOSE_	*Guardian Procedure Calls Reference Manual*
PROCESSHANDLE_GETMINE_	*Guardian Procedure Calls Reference Manual*
PROCESSHANDLE_NULLIT_	*Guardian Procedure Calls Reference Manual*
PROCESSHANDLE_TO_CRTPID_	*Guardian Procedure Calls Reference Manual*
PROCESSHANDLE_TO_FILENAME_	*Guardian Procedure Calls Reference Manual*
PROCESSHANDLE_TO_STRING_	*Guardian Procedure Calls Reference Manual*
PROCESSINFO	*Guardian Procedure Calls Reference Manual*
PROCESSNAME_CREATE_	*Guardian Procedure Calls Reference Manual*
PROCESSOR_GETINFOLIST_	*Guardian Procedure Calls Reference Manual*
PROCESSOR_GETNAME_	*Guardian Procedure Calls Reference Manual*
PROCESSORSTATUS	*Guardian Procedure Calls Reference Manual*
PROCESSORTYPE	*Guardian Procedure Calls Reference Manual*
PROCESSSTRING_SCAN_	*Guardian Procedure Calls Reference Manual*
PROCESSTIME	*Guardian Procedure Calls Reference Manual*
PROGRAMFILENAME	*Guardian Procedure Calls Reference Manual*
PURGE	*Guardian Procedure Calls Reference Manual*
PUTPOOL	*Guardian Procedure Calls Reference Manual*
RAISE_	*Guardian Procedure Calls Reference Manual*
READ	*Guardian Procedure Calls Reference Manual*
READ^FILE	*Guardian Procedure Calls Reference Manual*
READEDIT	*Guardian Procedure Calls Reference Manual*
READEDITP	*Guardian Procedure Calls Reference Manual*
READLOCK	*Guardian Procedure Calls Reference Manual*
READLOCKX	*Guardian Procedure Calls Reference Manual*
READUPDATE	*Guardian Procedure Calls Reference Manual*
READUPDATELOCK	*Guardian Procedure Calls Reference Manual*
READUPDATELOCKX	*Guardian Procedure Calls Reference Manual*
READUPDATEX	*Guardian Procedure Calls Reference Manual*
READX	*Guardian Procedure Calls Reference Manual*
RECEIVEINFO	*Guardian Procedure Calls Reference Manual*

REFPARAM_BOUNDSCHECK_	*Guardian Procedure Calls Reference Manual*
REFRESH	*Guardian Procedure Calls Reference Manual*
REMOTEPROCESSORSTATUS	*Guardian Procedure Calls Reference Manual*
REMOTETOSVERSION	*Guardian Procedure Calls Reference Manual*
RENAME	*Guardian Procedure Calls Reference Manual*
REPLY	*Guardian Procedure Calls Reference Manual*
REPLYX	*Guardian Procedure Calls Reference Manual*
REPOSITION	*Guardian Procedure Calls Reference Manual*
RESERVELCBS	*Guardian Procedure Calls Reference Manual*
RESETSYNC	*Guardian Procedure Calls Reference Manual*
RESIZEPOOL	*Guardian Procedure Calls Reference Manual*
RESIZESEGMENT	*Guardian Procedure Calls Reference Manual*
RESUMETRANSACTION	*TMF Application Programmer's Guide*
SAVEPOSITION	*Guardian Procedure Calls Reference Manual*
SEGMENT_ALLOCATE_	*Guardian Procedure Calls Reference Manual*
SEGMENT_ALLOCATE_CHKPT_	*Guardian Procedure Calls Reference Manual*
SEGMENT_DEALLOCATE_	*Guardian Procedure Calls Reference Manual*
SEGMENT_DEALLOCATE_CHKPT_	*Guardian Procedure Calls Reference Manual*
SEGMENT_GETBACKUPINFO_	*Guardian Procedure Calls Reference Manual*
SEGMENT_GETINFO_	*Guardian Procedure Calls Reference Manual*
SEGMENT_USE_	*Guardian Procedure Calls Reference Manual*
SEGMENTSIZE	*Guardian Procedure Calls Reference Manual*
SENDBREAKMESSAGE	*Guardian Procedure Calls Reference Manual*
SERVERCLASS_DIALOG_ABORT_	*TS/MP Pathsend and Server Programming Manual*
SERVERCLASS_DIALOG_BEGIN_	*TS/MP Pathsend and Server Programming Manual*
SERVERCLASS_DIALOG_END_	*TS/MP Pathsend and Server Programming Manual*
SERVERCLASS_DIALOG_SEND_	*TS/MP Pathsend and Server Programming Manual*

SERVERCLASS_SEND_	*TS/MP Pathsend and Server Programming Manual*
SERVERCLASS_SEND_INFO_	*TS/MP Pathsend and Server Programming Manual*
SET^FILE	*Guardian Procedure Calls Reference Manual*
SETJMP_	*Guardian Procedure Calls Reference Manual*
SETMODE	*Guardian Procedure Calls Reference Manual*
SETMODENOWAIT	*Guardian Procedure Calls Reference Manual*
SETMYTERM	*Guardian Procedure Calls Reference Manual*
SETPARAM	*Guardian Procedure Calls Reference Manual*
SETSTOP	*Guardian Procedure Calls Reference Manual*
SETSYNCINFO	*Guardian Procedure Calls Reference Manual*
SETSYSTEMCLOCK	*Guardian Procedure Calls Reference Manual*
SHIFTSTRING	*Guardian Procedure Calls Reference Manual*
SIGACTION_	*Guardian Procedure Calls Reference Manual*
SIGACTION_INIT_	*Guardian Procedure Calls Reference Manual*
SIGACTION_RESTORE_	*Guardian Procedure Calls Reference Manual*
SIGACTION_SUPPLANT_	*Guardian Procedure Calls Reference Manual*
SIGADDSET_	*Guardian Procedure Calls Reference Manual*
SIGDELSET_	*Guardian Procedure Calls Reference Manual*
SIGEMPTYSET_	*Guardian Procedure Calls Reference Manual*
SIGFILLSET_	*Guardian Procedure Calls Reference Manual*
SIGISMEMBER_	*Guardian Procedure Calls Reference Manual*
SIGJMP_MASKSET_	*Guardian Procedure Calls Reference Manual*
SIGLONGJMP_	*Guardian Procedure Calls Reference Manual*
SIGNAL_	*Guardian Procedure Calls Reference Manual*
SIGNALPROCESSTIMEOUT	*Guardian Procedure Calls Reference Manual*
SIGNALTIMEOUT	*Guardian Procedure Calls Reference Manual*
SIGPENDING_	*Guardian Procedure Calls Reference Manual*
SIGPROCMASK_	*Guardian Procedure Calls Reference Manual*
SIGSETJMP_	*Guardian Procedure Calls Reference Manual*

SIGSUSPEND_	*Guardian Procedure Calls Reference Manual*
SORTBUILDPARM	*FastSort Manual*
SORTERROR	*FastSort Manual*
SORTERRORDETAIL	*FastSort Manual*
SORTERRORSUM	*FastSort Manual*
SORTMERGEFINISH	*FastSort Manual*
SORTMERGERECEIVE	*FastSort Manual*
SORTMERGESEND	*FastSort Manual*
SORTMERGESTART	*FastSort Manual*
SORTMERGESTATISTICS	*FastSort Manual*
SPI_BUFFER_FORMATFINISH_	*DSM Template Services Manual*
SPI_BUFFER_FORMATNEXT_	*DSM Template Services Manual*
SPI_BUFFER_FORMATSTART_	*DSM Template Services Manual*
SPI_FORMAT_CLOSE_	*DSM Template Services Manual*
SPOOLBATCHNAME	*Spooler Programmer's Guide*
SPOOLCONTROL	*Spooler Programmer's Guide*
SPOOLCONTROLBUF	*Spooler Programmer's Guide*
SPOOLEND	*Spooler Programmer's Guide*
SPOOLERCOMMAND	*Spooler Programmer's Guide*
SPOOLEREQUEST	*Spooler Programmer's Guide*
SPOOLEREQUEST2	*Spooler Programmer's Guide*
SPOOLERSTATUS	*Spooler Programmer's Guide*
SPOOLERSTATUS2	*Spooler Programmer's Guide*
SPOOLJOBNUM	*Spooler Programmer's Guide*
SPOOLSETMODE	*Spooler Programmer's Guide*
SPOOLSTART	*Spooler Programmer's Guide*
SPOOLWRITE	*Spooler Programmer's Guide*
SQLADDR	*SQL/MP Programming Manual for COBOL85*
SQLCA_DISPLAY2_	*SQL/MP Programming Manual for COBOL85*
SQLCA_TOBUFFER2_	*SQL/MP Programming Manual for COBOL85*

SQLCADISPLAY	*SQL/MP Programming Manual for C, SQL Programming Manual for TAL*
SQLCAFSCODE	*SQL/MP Programming Manual for COBOL85, SQL/MP Programming Manual for C, SQL Programming Manual for TAL*
SQLCAGETINFOLIST	*SQL/MP Programming Manual for C, SQL Programming Manual for TAL*
SQLCATOBUFFER	*SQL/MP Programming Manual for C, SQL Programming Manual for TAL*
SQLGETCATALOGVERSION	*SQL/MP Programming Manual for COBOL85, SQL/MP Programming Manual for C, SQL Programming Manual for TAL*
SQLGETOBJECTVERSION	*SQL/MP Programming Manual for COBOL85, SQL/MP Programming Manual for C, SQL Programming Manual for TAL*
SQLGETSYSTEMVERSION	*SQL/MP Programming Manual for COBOL85, SQL/MP Programming Manual for C, SQL Programming Manual for TAL*
SQLSADISPLAY	*SQL/MP Programming Manual for COBOL85, SQL/MP Programming Manual for C, SQL Programming Manual for TAL*
SSGET	*SPI Programming Manual*
SSGETTKN	*SPI Programming Manual*
SSIDTOTEXT	*SPI Programming Manual*
SSINIT	*SPI Programming Manual*
SSMOVE	*SPI Programming Manual*
SSMOVETKN	*SPI Programming Manual*
SSNULL	*SPI Programming Manual*
SSPUT	*SPI Programming Manual*
SSPUTTKN	*SPI Programming Manual*
STATUSTRANSACTION	*TMF Application Programmer's Guide*
STEPMOM	*Guardian Procedure Calls Reference Manual*
STOP	*Guardian Procedure Calls Reference Manual*
STRING_UPSHIFT_	*Guardian Procedure Calls Reference Manual*
SUSPENDPROCESS	*Guardian Procedure Calls Reference Manual*

SYSTEMENTRYPOINT_RISC_	*Guardian Procedure Calls Reference Manual*
SYSTEMENTRYPOINTLABEL	*Guardian Procedure Calls Reference Manual*
TAKE^BREAK	*Guardian Procedure Calls Reference Manual*
TEXTTOSSID	*Guardian Procedure Calls Reference Manual*
TEXTTOTRANSID	*TMF Application Programmer's Guide*
TMF_SETTXHANDLE_	*TMF Application Programmer's Guide*
TMF_SUSPEND_	*TMF Application Programmer's Guide*
TMF_TXBEGIN_	*TMF Application Programmer's Guide*
TMF_BEGINTAG_FROM_TXHANDLE_	*TMF Application Programmer's Guide*
TMF_GETTXHANDLE_	*TMF Application Programmer's Guide*
TMF_GET_TX_ID_	*TMF Application Programmer's Guide*
TMF_RESUME_	*TMF Application Programmer's Guide*
TMF_TXHANDLE_FROM_BEGINTAG_	*TMF Application Programmer's Guide*
TIMESTAMP	*TMF Application Programmer's Guide*
TOSVERSION	*Guardian Procedure Calls Reference Manual*
TIME	*Guardian Procedure Calls Reference Manual*
TIMER_START_	*Guardian Procedure Calls Reference Manual*
TIMER_STOP_	*Guardian Procedure Calls Reference Manual*
TRANSIDTOTEXT	*TMF Application Programmer's Guide*
TS_NANOSECS_	*Guardian Procedure Calls Reference Manual*
TS_UNIQUE_COMPARE_	*Guardian Procedure Calls Reference Manual*
TS_UNIQUE_CONVERT_TO_JULIAN_	*Guardian Procedure Calls Reference Manual*
TS_UNIQUE_CREATE_	*Guardian Procedure Calls Reference Manual*
UNLOCKFILE	*Guardian Procedure Calls Reference Manual*
UNLOCKREC	*Guardian Procedure Calls Reference Manual*
UNPACKEDIT	*Guardian Procedure Calls Reference Manual*
USER_AUTHENTICATE_	*Guardian Procedure Calls Reference Manual*
USER_GETINFO_	*Guardian Procedure Calls Reference Manual*
USER_GETNEXT_	*Guardian Procedure Calls Reference Manual*
USERDEFAULTS	*Guardian Procedure Calls Reference Manual*

USERIDTOUSERNAME	*Guardian Procedure Calls Reference Manual*
USERNAMETOUSERID	*Guardian Procedure Calls Reference Manual*
USESEGMENT	*Guardian Procedure Calls Reference Manual*
VERIFYUSER	*Guardian Procedure Calls Reference Manual*
VRO_SET_	*Guardian Procedure Calls Reference Manual*
WAIT^FILE	*Guardian Procedure Calls Reference Manual*
WRITE	*Guardian Procedure Calls Reference Manual*
WRITE^FILE	*Guardian Procedure Calls Reference Manual*
WRITEEDIT	*Guardian Procedure Calls Reference Manual*
WRITEEDITP	*Guardian Procedure Calls Reference Manual*
WRITEREAD	*Guardian Procedure Calls Reference Manual*
WRITEREADX	*Guardian Procedure Calls Reference Manual*
WRITEUPDATE	*Guardian Procedure Calls Reference Manual*
WRITEUPDATEUNLOCK	*Guardian Procedure Calls Reference Manual*
WRITEUPDATEUNLOCKX	*Guardian Procedure Calls Reference Manual*
WRITEUPDATEX	*Guardian Procedure Calls Reference Manual*
WRITEX	*Guardian Procedure Calls Reference Manual*
XBNDSTEST	*Guardian Procedure Calls Reference Manual*
XSTACKTEST	*Guardian Procedure Calls Reference Manual*

Using the DIVER and DELAY Programs

This appendix describes the DIVER and DELAY programs which you can use to supplement the testing of an application program that runs as a process pair. DIVER causes a specified processor to fail and then prepares the processor for reload. DIVER can be used with DELAY to cause repeated failures and reloads of the processors in a system. This failure cycle allows you to test an application for fault tolerance while the processors are being halted and reloaded.

Running the DIVER Program

You run the DIVER program in a processor selected to fail. DIVER stops the processor such that it no longer transmits its “I’m alive” message. Because the other processors in the system do not receive this message, they collectively declare that the processor has failed. The difference between a processor halt and a processor running the DIVER program is that a processor halt results in the report of a processor halt code and a potential system freeze; a processor running the DIVER program results in the report of the processor event message ZCPU-EVT-DIVER. For more information on this event message, see the *NonStop Kernel Event Management Programming Manual*.

- Note these cautions when running the DIVER program
  - When running the DIVER program, it is recommended that the processor run-option always be included. Otherwise, the processor in which DIVER is intended to run is chosen by the normal process creation rules, which might not place it in the processor that you intend to fail.
  - If there is a $CMON process running in the system, it might affect the choice of processor where DIVER will run. Unless you are sure that the $CMON process will not alter the processor specified in the DIVER command, stop $CMON before running DIVER.
  - Do not use DIVER to try to halt a processor in order to get a dump. When setting up the processor for a RELOAD command, DIVER destroys its memory contents. Consult the operations guide for your system type for details on taking dumps.
  - Do not run the DIVER program if the backup CPU for its I/O processes has been reloaded within the last five minutes. Otherwise, a CPU halt may occur.

After stopping the processor, DIVER sets up the processor for a RELOAD command, as if a processor reset and load operation occurred. You can then reload the processor.

Before DIVER brings down a processor, it verifies that the requester's process access ID (PAID) is a member of the super group.

If the requester's PAID is not a member of the super group, DIVER abends and the processor continues to be operational.

The syntax to run DIVER is:

```
DIVER / run-option [ , run-option ] ... /
```

DIVER

is an implicit RUN command that starts a DIVER process.

run-option

is any valid option for the TACL RUN command. These options are described in the *TACL Reference Manual*.

Running the DELAY Program

You use the DELAY program to delay the execution of the calling process for a specified amount of time. DELAY calls the DELAY procedure for the length of time specified. After the delay finishes, the process resumes execution.

The syntax to run DELAY is:

```
DELAY / run-option [ , run-option ].../time { TIC | SEC | MIN
}
```

DELAY

is an implicit RUN command that starts a DELAY process.

run-option

is any valid option for the TACL RUN command. These options are described in the *TACL Reference Manual*.

time

is an integer specifying the delay time in the specified units.

{ TIC | SEC | MIN }

specifies the units of measurement for *time*:

TIC hundredths of a second

SEC	seconds
MIN	minutes

Example Using DIVER and DELAY

This example uses the DIVER and DELAY programs to cause both processors (0 and 1) of a two-processor system to fail and reload repeatedly, approximately once every seven minutes. You can increase this cycle time if your application requires more time to recover.

Note. If your test system has more than two processors, then examine the system configuration and select processors to fail that will cause the least disruption to the system.

1. Start the TACL command interpreter running as a process pair in processors 0 and 1. In this example, processor 0 must be the primary processor, and processor 1 must be the backup processor for the TACL process.
2. Log on as a member of the super group.
3. Create two EDIT files (file code 101) named DIVE0 and DIVE1 that contain this TACL commands.

   The DIVE0 file contains:

```
DIVER / CPU 1 /
DELAY 120 SEC
RELOAD 1
DELAY 5 MIN
TACL / CPU 1, NAME $DIVE1, IN DIVE1 /
```

   The DIVE1 file contains:

```
DIVER / CPU 0 /
DELAY 120 SEC
RELOAD 0
DELAY 5 MIN
TACL / CPU 0, NAME $DIVE0, IN DIVE0 /
```

4. Start your application running as a process pair in processors 0 and 1.
5. Start the processor failure cycle by entering this command at the TACL prompt. This command starts a TACL process using DIVE0 as the IN file.

```
> TACL /CPU 0, NAME $DIVE0, IN DIVE0, OUT $0 /
```

6. To stop the processor failure cycle, enter:

```
> STOP $DIVE0
> STOP $DIVE1
```

J System Limits

This appendix presents a series of tables that summarize the architectural and programmatic limits that apply on NonStop servers. For SQL/MP limits, see the "Limits" entry in the *SQL/MP Reference Manual*. For TMF limits, see the *TMF Configuration and Planning Guide.* For the locations of other published limits, see Table J-7 on page J-10.

Table J-1. System-Level Limits (page 1 of 2)

Limit Description	Maximum Value	Comment
Processors per node	16	
Nodes per Expand network	255	
Direct neighbors per node	63	
Nodes per Fox ring	14	G-series only
OSIMAGE size	128 MB	OSIMAGE must be on the system-load volume. The H-series OSIMAGE is much smaller, because the libraries and programs that were formerly in OSIMAGE are now in separate object files.
Number of parameters to a procedure	32	TAL limit. The maximum is 32 for a callable function and 31 if procedure is callable and variable or extensible. Parameters must also fit in 64 mask bits. For C/C++ functions that are not variable or extensible, there is effectively no limit.
Number of characters in procedure name in TAL or pTAL	31	There is no specific limit for identifiers in C/C++, but function names longer than 32 characters cannot be exported from the system library.
Number of entries in destination control table (DCT)	65,376	Devices share DCT with named processes.
Logical device numbers (LDEVs)	65,376	
Device types	64K	
Device/process subtypes	64	
Number of subdevices per device	Varies by subsystem	See appropriate subsystem manual.

Table J-1. System-Level Limits (page 2 of 2)

Limit Description	Maximum Value	Comment
Number of tape drives per node in a labeled tape environment	20	
System-generated names (default, 4-character form)	30,720	Names in the form $X*naa*, $Y*naa*, $Z*naa*, where *a* is alphanumeric and *n* is numeric
System-generated names (5-character form)	30,000	Names in the form $X*nnnn*, $Y*nnnn*, $Z*nnnn*
Explicitly reserved process names	--	$X..., $Y..., $Z..., $0, $AOPR, $CMON, $CMP, $DM*nn* (*n* is numeric), $IMON, $IPB, $KEYS, $MLOK, $NCP, $NULL, $OSP, $PM, $S, $SPLP, $SPLS, $SSCP, $SYSTEM, $T, $TMP, $TRPM, $TRMS, $TSCH

Table J-2. Per-Process Limits (page 1 of 2)

Limit Description	Maximum Value	Comment
Segment IDs (SEGIDs)	Depends on whether the segment is privileged, nonprivileged, assigned or unassigned.	0 - 1023; reserved for customers; nonprivileged 1024 - 2047; assigned by HP; nonprivileged 2048 - 4095; assigned by HP; privileged (in two ranges) 4096 -65425; unassigned; privileged 65426 -65535; reserved internally, used for special classes of segments
Selectable segment size	127.5 MB	Before G05, the number of selectable segments per process is bounded by available aliased virtual memory. Starting with G05, selectable segments default to unaliased.
Size of flat segment area	480 MB	Up to 15 segments aligned on 32 MB (region) boundaries; maximum of 128 MB per segment (G04.00 and earlier G-series)

* All G-series RVUs and H06.05 and earlier H-series RVUs
** H06.06 and later H-series RVUs; all J-series RVUs
***H06.17 and later H-series RVUs; J06.06 and later J-series RVUs

Table J-2. Per-Process Limits (page 2 of 2)

Limit Description	Maximum Value	Comment
	1120 MB	G05.00 and all subsequent G-series RVUs
	1.5 GB	H06.01 and all subsequent H-series RVUs
Open user semaphores	64	
	8192***	
Concurrent outstanding messages	4095 *	The default limits are 255 incoming messages and 1,023 outgoing messages. These limits can be modified separately through the CONTROLMESSAGESYSTEM procedure.
	16383 **	
Time Slice	2 seconds for G-series RVUs	The time slice is the amount of time a process can run at a given priority before it is declared CPU-bound and has its priority decremented.
	400 milliseconds for H-series RVUs	This value is for NSE-A processors. This value could decrease on faster processors.

* All G-series RVUs and H06.05 and earlier H-series RVUs
** H06.06 and later H-series RVUs; all J-series RVUs
***H06.17 and later H-series RVUs; J06.06 and later J-series RVUs

Table J-3. Per-Processor Limits (page 1 of 3)

Limit Description	Maximum Value	Comment
Processes	64K	Architectural limit; current practical limit is much lower.
Processes	4000 for TNS/R when they are TNS processes, and 8000 for TNS/E total	Implementation limit. Limit is reached when processor runs out of a resource such as CPU cycles, types of memory (operating system aliased, memory, other virtual memory, or physical memory) or other limits documented in this Appendix.
Time-list elements (TLEs)	3,600 for TNS/R and 20000 for TNS/E	Limited by system data space availability; defaults to 1.5 times number of configured PCBs. The last 100 TLEs are available only to system processes.

Table J-3. Per-Processor Limits (page 2 of 3)

Timer tick value	10 millisecs for TNS/R and 1 microsecs for TNS/E	For example, see parameter to PROCESS_DELAY_ procedure.
Timestamp resolution	1 microsec	
Physical memory	Varies with server model	
Aliased virtual memory	< 1 GB	8,189 unit segments, where 1 unit segment = 128 KB (G04.00 and earlier G-series RVUs) 7,933 unit segments, where 1 unit segment = 128 KB (G05.00 and later G-series RVUs)
SYSPOOL size	32 KB	
EXTPOOL size	512 KB	
FLEXPOOL size	998 MB	Varies based on demand; limited by physical and aliased virtual memory.
Open user semaphores	Number of processes 65536*****	
Counters displayed by PEEK	32 or 64 bits	
Number of concurrent process creations and deletions	96	Number of Phoenix processes
Number of concurrent swap file opens	32K	
Absolute maximum number of message quick cell (MQCs)	<256K	Controls the absolute maximum number of concurrent outstanding messages to/from the processor

Table J-3. Per-Processor Limits (page 3 of 3)

Maximum memory allowed for MQCs	6.1 MB* 128 MB** 1GB ******	The maximum number of MQCs allowed per MQCs processor is the lesser of the following: (a) the absolute maximum, and (b) the maximum number of MQCs that can be instantiated given the maximum memory allowed for MQCs. Note that (b) depends on the MQC sizes instantiated in the processor. To view the MQC usage statistics in the processor, use the PEEK /CPU N/ MQCINFO command
Maximum MQC size	2,048 bytes*** 8,128 bytes****	Higher messaging performance is achieved when a message request and/or reply is cached in an MQC. Larger MQC sizes support a wider range of message sizes that can benefit from the MQC caching optimization.

* G06.24 and earlier G-series RVUs
** G06.25 and later G-series RVUs; all H-series and J-series RVUs.
*** All G-series RVUs; H06.06 and earlier RVUs
**** H06.07 and later H-series RVUs; all J-series RVUs
*****H06.17 and later H-series RVUs and J06.06 and later J-series RVUs
******H06.20 and later H-Series RVUs and J06.09 and later J-series RVUs

Table J-4. TNS vs. Native limits (page 1 of 2)

Limit Description	Maximum Value	Comment
TNS procedures per code segment	510	PEP size; code segment = 128 KB
Unique external procedures called from TNS program, per code segment	512	XEP size
TNS procedure size	64 KB	Practical limit
TNS user code and user library	32 segments	
TNS total stack and globals	64 KB	
TNS total user data space	128 KB	Does not include extended segments data space.
Accelerated user code, user library, or system code	28 MB each	

Table J-4. TNS vs. Native limits (page 2 of 2)

Native user code	32 MB on TNS and TNS/R systems, 256 MB on TNS/E systems	
Native user library	32 MB on TNS and TNS/R systems, 256 MB on TNS/E systems	
Native process SRLs	128	G-series RVUs
Native process DLLs	1000 on TNS/R systems, 700 on TNS/E systems	H-series RVUs. The native process DLLs do not have to be contiguous; DLLs and flat segments can be interspersed in the flat-segment area.
Native stack	32 MB	Default = 1 MB, H-series Default = 2MB
Native heap and globals	128 MB	Before G05
	1.1 GB	G05 RVU
	1.5 GB	H-series RVUs
Native priv stack	128 KB	This value represents the default size. By calling the HEADROOM_ENSURE_ procedure you can increase the size in H-series systems up to 880KB.
Native globals + heap + flat segments + stack	640 MB	Before G05
	1.1 GB	G05 RVUs
	1.5 GB	H-series RVUs

Table J-5. Enscribe File System Limits (page 1 of 2)

Limit Description	Maximum Value	Comment
File codes	64K	Reserved range is limited to 0 through 999.
File error numbers	64K	Errors above 255 are problematic through some old interfaces.
Setmodes	64K	
Controls	64K	
Setparams	64K	
System message numbers	32K	Restricted to negative values.
Concurrent opens	H-series and J-series	16360
	G-series	32720
Nowait depth	Varies by device type	See appropriate subsystem manual.
Sync depth for nonretryable writes to disk between checkpoints	15	
PFS Size	32 MB	H-series RVUs
	8 MB	G-series RVUs
Receive depth	16300******	Size of $RECEIVE file queue
Transactions per TFILE	1000	1 TFILE per process
Partitions per file	16	
Structured disk file transfer sizes (default mode):		
Opened with structured access	1 record	
Opened with unstructured access	4 KB	
Unstructured file transfer sizes (default mode):		
Audited file write transfer limitation	4 KB	
Remote across Expand	~32 KB or 56 KB	Depending upon Expand

******H06.18 and later H-series RVUs and J06.07 and later J-series RVUs

Table J-5. Enscribe File System Limits (page 2 of 2)

Local access/remote across SuperCluster	56 KB	With a 4 KB unstructured buffer size
Disk file transfer sizes using SETMODE 141		See SETMODE 141 (in SETMODE Functions on page 14-63) for restrictions
Remote across Expand	~32 KB or 56 KB	Depending upon Expand
Local access/remote across SuperCluster	56 KB	
Interprocess ($RECEIVE) transfer sizes:		
Remote across Expand	~32 KB or 56 KB	Depending upon Expand
Local access/remote across SuperCluster	56 KB	
Tape and other device transfer sizes:		Device-dependent
Remote across Expand	~32 KB or 56 KB	Depending upon Expand
Local access/remote across SuperCluster	56 KB	

Limit Description	**Format 1 Files Maximum Value**	**Format 2 Files Maximum Value**
Disk file partition size	2 GB minus 1 MB	1 TB
Single-partitioned disk file size	2 GB minus 1 MB	1 TB
Multipartitioned disk file sizes (key-sequenced)	16 * (2 GB minus 1 MB)	16 TB
Multipartitioned disk file sizes (not key-sequenced)	4 GB minus 4 KB	1 TB
Structured disk file record sizes:		
Entry-sequenced	4 KB - 24	4 KB - 48
Key-sequenced	4 KB - 34	4 KB - 56
Relative	4 KB - 24	4 KB - 48
Number of records per structured block	511	2 billion

******H06.18 and later H-series RVUs and J06.07 and later J-series RVUs

Table J-6. DP2 Limits

Limit Description	Maximum Value	Comment
Volumes per physical disk	1	Can be mirrored.
Structured disk block size	4 KB	
Unstructured disk file buffer size	4 KB	
Number of extents per file	<= 978	Depends on number of partitions and alternate keys specified.
Disk volume size	600 GB	
Cache per volume	1 GB	Cache is limited to the available physical and virtual memory space in the processor. Configuring a cache to be too large can cause memory pressure in the processor, which results in increased processor overhead in DP2 processes and can result in longer execution times for other applications. In extreme cases of memory pressure, an %11500 processor halt can occur.
Bulk I/O transfer	56 KB bytes	
Number of locks	5,000 per file open and per transaction	Lock limits can be raised by using the SYSGEN modifiers MAXLOCKSPEROCB and MAXLOCKSPERTCB. Maximum value for these limits is 100,000. See the *System Generation Manual for Disk and Tape Devices*. Configuring large values for these limits increases DP2 memory requirements and can affect DP2 response time.
Concurrent disk file opens	65, 279*	
Files per volume	32,763	

* H06.15 and later H-series RVUs and J06.04 and later J-series RVUs.

Table J-7. Other Published Limits

Manual Title	Location in Manual
COBOL Manual for TNS and TNS/R Programs	Section 20, “HP COBOL85 Limits”
COBOL Manual for TNS/E Programs	Section 20, “HP COBOL85 Limits”
NonStop TUXEDO System Administration Guide	Section 11, “NonStop TUXEDO Configuration Limits and Defaults”
FastSort Manual	Appendix E, “FastSort Limits”
Flow Map Manual	Appendix F, “FMH Internal Table Limits”
FORTRAN Reference Manual	Appendix E, “Compiler Limits”
SQL/MP Reference Manual	“Limits” entry
TMF Configuration and Planning Guide	Throughout manual
TS/MP Pathsend and Server Programming Manual	Appendix A, “NonStop TS/MP Limits for Pathsend Requesters”
TMF System Management Manual	Appendix C, “Configuration Limits and Defaults”
Virtual Hometerm Subsystem (VHS) Manual	Appendix B, “VHS Limits”
Pathway/iTS System Management Manual	Appendix C, “Configuration Limits and Defaults”
Pathway/Pathway/XM System Management Manual	Appendix C, “Configuration Limits and Defaults”
PS TEXT FORMAT Reference Manual	Appendix C, “Limits and Defaults”
Spooler Utilities Reference Manual	Appendix G, “Spooler Limits”
Surveyor Reference Manual	Appendix D, “The 128-Column Limit”

K Character Set Translation

Table K-1 contains an ASCII-EBCDIC translation. The sum of the hexadecimal row/column headings is the EBCDIC value corresponding to the ASCII value in the body of the table. Translation is symmetric; translating the contents of any array from ASCII to EBCDIC and back, or vice versa, returns the original text.

There is no recognized standard for EBCDIC, and several variations exist. The mapping of Table K-1 is consistent with that in other HP products.

Table K-1. Character Set Translation

	00	10	20	30	40	50	60	70	80	90	A0	B0	C0	D0	E0	F0
00	00	10	80	90		&	-	BA	C3	CA	D1	D8	{	}	\	0
01	01	11	81	91	A0	A9	/	BB	a	j	~	D9	A	J	9F	1
02	02	12	82	16	A1	AA	B2	BC	b	k	s	DA	B	K	S	2
03	03	13	83	93	A2	AB	B3	BD	c	l	t	DB	C	L	T	3
04	9C	9D	84	94	A3	AC	B4	BE	d	m	u	DC	D	M	U	4
05	09	85	0A	95	A4	AD	B5	BF	e	n	v	DD	E	N	V	5
06	86	08	17	96	A5	AE	B6	C0	f	o	w	DE	F	O	W	6
07	7F	87	1B	04	A6	AF	B7	C1	g	p	x	DF	G	P	X	7
08	97	18	88	98	A7	B0	B8	C2	h	q	y	E0	H	Q	Y	8
09	8D	19	89	99	A8	B1	B9	`	i	r	z	E1	I	R	Z	9
0A	8E	92	8A	9A	[	]	|	:	C4	CB	D2	E2	E8	EE	F4	FA
0B	0B	8F	8B	9B	.	$	,	#	C5	CC	D3	E3	E9	EF	F5	FB
0C	0C	1C	8C	14	<	*	%	@	C6	CD	D4	E4	EA	F0	F6	FC
0D	0D	1D	05	15	(	)	_	'	C7	CE	D5	E5	EB	F1	F7	FD
0E	0E	1E	06	9E	+	;	>	=	C8	CF	D6	E6	EC	F2	F8	FE
0F	0F	1F	07	1A	!	^	?	"	C9	D0	D7	E7	ED	F3	F9	FF

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Special Characters