HP NonStop TACL Reference Manual

Abstract

This publication describes the syntax and use of the HP Tandem Advanced Command Language (TACL) variables, commands, and built-in functions.

Product Version

T9205D46, T9205H01

Supported Release Version Updates (RVUs)

This publication supports J06.03 and all subsequent J-series RVUs, H06.03 and all subsequent H-series RVUs, G06.20 and all subsequent G-series RVUs, and D46.00 and all subsequent D-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
429513-017	August 2013

Document History

Part Number	Product Version	Published
429513-013	T9205D46, T9205H01	November 2010
429513-014	T9205D46, T9205H01	August 2011
429513-015	T9205D46, T9205H01	February 2012
429513-016	T9205D46, T9205H01	August 2012
429513-017	T9205D46, T9205H01	August 2013

Legal Notices

Printed in the US

HP NonStop TACL Reference Manual

Glossary | Index | Figures | Tables

Figures

Tables

What's New in This Manual

Manual Information

Abstract

This publication describes the syntax and use of the HP Tandem Advanced Command Language (TACL) variables, commands, and built-in functions.

Product Version

T9205D46, T9205H01

Supported Release Version Updates (RVUs)

This publication supports J06.03 and all subsequent J-series RVUs, H06.03 and all subsequent H-series RVUs, G06.20 and all subsequent G-series RVUs, and D46.00 and all subsequent D-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
429513-017	August 2013

Document History

Part Number	Product Version	Published
429513-013	T9205D46, T9205H01	November 2010
429513-014	T9205D46, T9205H01	August 2011
429513-015	T9205D46, T9205H01	February 2012
429513-016	T9205D46, T9205H01	August 2012
429513-017	T9205D46, T9205H01	August 2013

New and Changed Information

Changes to the 429513-017 manual

- Modified the description of the STOP option on page 8-208.
- Added the FORCED option and its description on page 8-208.
- Added a confirmation message below the command line on page 8-214.
- Added more modes in the table and modified the information on modes under mode on page 9-359.
- Modified the information on error results under Result on page 9-359.

- Added more information on customization of TACLCSTM and TACLLOCL under Customizing the TACL Environment.
- Added the IPUASSOCIATION, IPUNUMBER, PROCESSCREATIONTIME, and PROGRAMDATAMODEL options under #PROCESSINFO Built-In Function on pages 9-295, 9-297, 9-299.
- Added new information on the output of the TACL STATUS command along with the DETAIL option under Considerations on page 8-210.
- Added a new section Installation Instructions on page 6-1.
- Added three states of an ancestor and their description under aid on page 8-127.
- Added a note on subvolume names and #PMSEARCHLIST on page 9-286.
- Modified STATUS Command on page 8-206.

Changes to the 429513-016 manual

- Updated the IPUCOM Program Considerations on page 88.
- Updated the RUN[D|V] Command on page 8-156.
- Added SEMSTAT Program on page 8-169.

Changes to the 429513-015 manual

- Added CLICVAL program and its function in Commands and Programs table.

Changes to the 429513-014 manual

- Added IPUCOM program and its brief function in Commands and Programs table.
- Added the detailed description of IPUCOM Program in chapter 8.
- Updated Considerations section for FILEINFO Command on page 8-68.

Changes to the 429513-013 manual

- Added reference to the *Guardian Procedure Calls Reference* manual for a list of processor types and the associated number under Results on page 9-310.
- Removed outdated table listing processors and their associated numbers from 9-310.
- Reworded information under Result on page 9-309.
- Updated error message for the `FILEINFO` command on page 8-68.

About This Manual

This manual contains reference material describing the HP Tandem Advanced Command Language (TACL). It presents the syntax and operations of all standard commands and functions available in the :UTILS:TACL directory (embodying the command interpreter capability of the TACL product), as well as the syntax and description of TACL built-in functions and built-in variables (the programming language aspect of the TACL product).

Audience

This manual is intended for all users of the TACL product, both users who intend to use TACL primarily as a command interpreter, and those users who need to use the extensible capabilities of TACL for programming. Also included in this manual are descriptions of TACL operations that are restricted for use by system management personnel only.

Organization

Section 1, Overview of TACL	Provides an overview of TACL features and operation.
Section 2, Lexical Elements	Describes the basic elements of the TACL product.
Section 3, Expressions	Describes syntax and semantics for expressions.
Section 4, Variables	Describes variable types and their uses.
Section 5, Statements and Programs	Describes syntax and semantics for TACL statements.
Section 6, The TACL Environment	Describes environmental information about the TACL product, including required and recommended files, initialization, communication with a CMON process, defining library files, and defining and using segment files and directories.
Section 7, Summary of Commands and Built-In Functions	Lists commands and built-in functions by functional group.
Section 8, UTILS:TACL Commands and Functions	Describes commands and functions provided with the TACL product in the :UTILS:TACL directory.
Section 9, Built-In Functions and Variables	Describes TACL built-in functions and variables.

Appendix A, Syntax Summary	Provides a syntax summary of all TACL functions.
Appendix B, Error Messages	Describes types of error messages and includes a list of TACL error messages.
Appendix C, Mapping TACL Built-In Functions to Guardian Procedures	Maps TACL built-in functions to Guardian procedures.

Related Reading

The following paragraphs list manuals that are related to the use of the TACL product, either as a command interpreter or as a programming language.

Prerequisites

The *Guardian User's Guide* presents introductory material about the TACL product, including:

- Use of TACL as a command interpreter
- Defining function keys
- Writing simple macros
- Using DEFINEs

To gain a basic understanding of the use of TACL, read the first four sections of the *Guardian User's Guide* before using this manual. You should also be familiar with basic programming concepts and terminology, such as "pushing" and "popping" (creating and deleting) variables, using arguments, and so on.

Corequisites

If you use TACL as a programming language to create macros and routines, the *TACL Programming Guide* presents task-oriented material and examples. (The *TACL Programming Guide* does not, however, contain descriptions of restricted commands and functions.) Additional sources of information depend on your use of the TACL product. Manuals of possible interest include:

- Manuals about utilities, such as the *File Utility Program (FUP) Reference Manual*.
- Manuals about debugging programs in languages other than TACL (that you run from TACL), such as the *Debug Manual* and the *Inspect Manual*.
- Manuals about Distributed Systems Management (DSM), including:
  - EMS Reference Summary for reference information about EMS
- Manuals about the HP NonStop™ operating system, including:

- *Guardian Programmer's Guide*
- *Guardian Procedure Calls Reference Manual*
- *Guardian Procedure Errors and Messages Manual*
- *NonStop S-Series Operations Guide*
- *Security Management Guide*

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 edition of the manual and the preceding edition. 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 HP 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 you have concerning this document.

1 Overview of TACL

TACL is a software application that provides an interface to the HP NonStop operating system. You can use TACL either as an interactive command interface or as an interpreted programming language to construct programs.

This manual describes the syntactical elements of TACL.

- The *Guardian User's Guide* describes how to use TACL interactively.
- The *TACL Programming Guide* describes how to construct TACL programs.

As a programming language, the TACL product is most often used for managing systems and processes. You can, for example, use TACL to:

- Automate system startup and shutdown procedures
- Automate subsystem startup and shutdown procedures
- Run utilities and issue commands either with a fixed set of commands or a flexible set that you can tailor at run time
- Create a customized environment that simplifies commonly performed tasks for users
- Control subsystem operation using the Subsystem Programmatic Interface (SPI)
- Communicate with the Event Management Service (EMS) and generate EMS messages

The TACL language consists of commands, built-in functions, and built-in variables. Commands are typically used for interactive work. Built-in functions are typically used for programmatic work. Built-in variables store environmental information; you can set and retrieve their values.

Procedural constructs such as flow control statements are provided as part of the set of built-in functions. In addition, TACL provides powerful text manipulation functions that process output and results from processes.

TACL is extensible. Consult the documentation that accompanies each software RVU to determine if additional function has been added.

Using TACL Interactively

After you log on to a TACL process, TACL provides an environment that includes predefined commands such as:

RUN	Runs a process
STATUS	Displays information about one or more running processes
ENV and WHO	Describes your TACL environment

Your environment can include user-defined commands if you or your system manager define them.

To start a process (such as FUP) or use a TACL command (such as TIME), type the program name; this is known as invoking the program or the command. If you type a command incorrectly, TACL issues an error message and includes the expected syntax. You can then retype the command.

To obtain syntax help while you are typing a command, press the F16 key (or the appropriate help key defined in your environment) at the point in the command where you want help.

The *Guardian User's Guide* provides detailed instructions for using TACL interactively, including:

- Logging on and logging off
- Obtaining information about system users
- Editing command lines
- Managing files
- Starting and controlling processes
- Defining function keys
- Creating and using DEFINEs

The use of commands is the simplest interactive use of TACL. You can, however, access TACL built-in variables and functions or run your own procedural constructs interactively, as follows:

- Display the contents of TACL built-in variables by typing the variable name (including the initial number sign).
- Invoke a TACL built-in function by typing the function name enclosed in square brackets.

(First, set the #INFORMAT value to TACL as described in #INFORMAT Built-In Variable on page 9-196. Otherwise, TACL will not recognize the square brackets as special characters.)

To define and use your own variables, you must understand the use of square brackets and other lexical elements; For additional information, see Section 2, Lexical Elements, and Section 4, Variables.

Developing TACL Programs

The development of TACL programs is a more advanced use of TACL; programs are useful if you plan to run the same set of TACL commands or built-in functions frequently.

TACL allows you to interact with processes, handle results, and make decisions about further actions; it provides string-handling capabilities, character-handling capabilities, exception-handling capabilities, and many built-in functions that provide information about the system environment. You can use TACL to interface with the Subsystem Programmatic Interface (SPI), the Event Management Service (EMS), and other utilities and programs.

This manual describes TACL syntax and semantics. For more information about how to use TACL for specific programming tasks, see the *TACL Programming Guide*.

Language Features

TACL provides a set of predefined commands, functions, and data variables. These entities are built into the TACL language and are the building blocks with which you construct TACL programs:

- TACL built-in functions are intended for use in TACL programs. Built-in function names start with a number sign (#). Most functions return a result that can be analyzed programmatically. Examples include:

#ARGUMENT	Parses the arguments passed to a routine
#INPUT	Reads information from the TACL IN file
#PROCESSINFO	Returns information about a process

  Built-in functions also provide flow control, such as loop control and exit mechanisms. To see a list of functions, use the #BUILTINS built-in function or see Section 9, Built-In Functions and Variables.

- TACL built-in variables contain information about the TACL environment. Built-in variable names start with a number sign (#). These variables are used primarily to establish the TACL environment. You can set and retrieve their values, and create new instances of these variables, but you cannot delete the variables themselves. Examples of these variables include:

#OUT	The name of the OUT file used by TACL
#PMSG	The state of the PMSG flag When set, TACL displays a message whenever processes associated with your TACL process start and stop.
#MYTERM	The name of the home terminal

- TACL commands (such as RUN and STOP) are intended for interactive use, but you can use them in TACL programs. Commands do not return status or error information; they usually display results.

  TACL provides these procedural features:

  - Data structures. TACL supports text and STRUCT variables.

- Data types. TACL interprets text variables as text unless you request an arithmetic or logical operation. TACL supports an extensive set of data types for STRUCT variables and for arguments to routine variables. Programs. TACL programs allow you to define a block of TACL statements that performs one or more tasks. You can access a TACL program from other TACL programs; you can nest programs within other programs. For a description of TACL programs, see Section 5, Statements and Programs.
- Argument passing. You can pass arguments to TACL programs. The mechanism depends on the type of program (text, macro, or routine).
- Data operations. You can compare, move, and manipulate the contents of variables that contain text or TACL statements.

For additional information about TACL programs, see Section 5, Statements and Programs.

Program Development Tools

TACL provides a symbolic debugger that gives you interactive debugging capabilities such as breakpoints and step operations. For additional information about the debugger, see the _DEBUGGER Command in Section 8, UTILS:TACL Commands and Functions, or the *TACL Programming Guide*.

Using TACL With Other Subsystems

TACL can communicate with other programs. This manual describes each command and function that supports network and interprocess communication. For examples showing the use of the commands and functions and for information about subsystem-specific information such as SPI tokens supported by TACL, see the *TACL Programming Guide*.

2 Lexical Elements

TACL consists of these lexical elements:

- Character Set
- Special Characters
- Constants
- Reserved Words
- Comments

Character Set

TACL supports the ISO 8859.1 (International Organization for Standardization) character set, also called the ECMA-94 (European Computer Manufacturers Association) character set, which is an 8-bit character set with 256 character positions. The first 128 characters are the same as the ASCII (American Standard Code for Information Interchange) character set; ISO 8859.1 is a superset of ASCII.

The high-order 128 positions include 94 characters that are used in the majority of western European languages, plus two additional formatting characters: a nonbreaking space and an optional or discretionary hyphen.

Data

You can use all characters in the ISO character set as data.

Variable Names

You can use all printable characters in TACL variable names; however, you can use only the low-order (ASCII) characters in the names of systems, disk volumes, subvolumes, files, processes, and devices.

Upshifting

TACL automatically upshifts variable names when it defines them. If a variable name contains lowercase letters that contain diacritical marks (marks added to a letter to indicate a special phonetic value), those characters may lose their diacritical marks when upshifted using CPRULES0. This upshifting can change the apparent identity of the variable. To avoid errors, use uppercase letters if your variable name includes diacritical marks.

TACL provides two files that contain coded rules for upshifting characters:

- CPRULES0 contains the rules used by the majority of western European countries; this is the default set of character-processing rules.
- CPRULES1 contains the rules used primarily by Spain and French Canada.

Special Characters

There are six types of characters that have special meaning to TACL:

Character Name	Description
Metacharacters	Requests that TACL interpret subsequent text in a special way
Separator	Delineates keywords, variable names, and so forth
Question mark	Command line editing directive
Ampersand	Continues current line on the next physical line
Template	Provides wild-card matching for file names
Operator	Specifies arithmetic, relational, or logical operations

Metacharacters

The interpretation of special characters as metacharacters depends on the setting of the #INFORMAT Built-In Variable on page 9-196. The display of special characters depends on the setting of the #OUTFORMAT Built-In Variable on page 9-274.

The #INFORMAT built-in variable affects the interpretation of special characters read from the IN file, including terminal input and files read by the OBEY command. #INFORMAT can have one of three values:

- TACL: Metacharacters have full effect.
- PLAIN: Metacharacters are treated as ordinary characters.
- QUOTED: If metacharacters are contained within quotation marks (“ ”), they are treated as ordinary text.

?TACL MACRO files, ?TACL ROUTINE files, and library files read by LOAD or #LOAD are read in TACL format unless they contain ?FORMAT directives that specify PLAIN or QUOTED format. For more information about the ?FORMAT directive, see Section 5, Statements and Programs.

Note. TACL treats metacharacters as PLAIN when they are transmitted by #REQUESTER or #SERVER, so characters obtained from these functions do not retain their special properties. For example, square brackets in text obtained from a file opened by #REQUESTER do not cause enclosed text to be expanded. This also applies to FILETOVAR and VARTOFILE, which use the #REQUESTER built-in function.

Table 2-1 lists TACL metacharacters.

Table 2-1. TACL Metacharacters

Character(s)	Name	Description
[ ]	Square brackets	Causes TACL to expand the enclosed text For additional information, see Square Brackets ([ ]) on page 2-3.
==	Double equal signs	Specifies a comment from the equal signs to the end of the line For additional information, see Comments on page 2-10.
{ }	Pairs of braces	Specifies a label; used in #CASE, #DEF, #IF, and #LOOP functions For example, |THEN| and |ELSE| are labels. For more information about enclosures and labels, see Function Calls on page 5-1.
| |	Pair of vertical lines	Specifies a label; used in #CASE, #DEF, #IF, and #LOOP functions For example, |THEN| and |ELSE| are labels. For more information about enclosures and labels, see Function Calls on page 5-1.
~	Tilde	Changes the interpretation of the next character (or in the case of double equal signs two characters) For additional information, see Tilde (~) on page 2-5.

Square Brackets ([ ])

You can use square brackets in several ways:

- To extend a logical line past the physical line limit. If you enclose more than one line within a pair of square brackets, the brackets mark the beginning and ending of the logical line. You can extend the line to a maximum of 239 characters.
- To define an enclosure that contains a label and one or more TACL statements. Enclosures can be used in #CASE, #DEF, #IF, and #LOOP built-in functions. For more information about enclosures and labels, see Function Calls on page 5-1.
- To specify that TACL expand a variable name to the contents of the variable. When TACL expands a variable name, it replaces the variable name with the contents of the variable. To expand the variable, enclose the name of the variable in square brackets. For information about variables, see Section 4, Variables.

  If the variable contains text, TACL replaces the bracketed variable name with the text stored in the variable. For example, if the variable var1 contains the integer 3,

you can enclose var1 in square brackets to obtain the contents of var1 or you can omit the brackets to obtain the text var1:

```
12> #OUTPUT [var1]
3
13> #OUTPUT var1
var1
14>
```

If the variable is a routine that contains executable TACL statements, TACL interprets the executable statements and replaces the variable name with the result of the statements.

When specifying the contents of a variable as an argument to a command or built-in function, you do not always need to enclose the variable name in square brackets:

- Commands and built-in functions such as VCHANGE and #CHARGETV expect a variable name for one or more arguments. When you supply a variable name as an argument to one of these commands or built-in functions, you do not need to enclose a variable name in square brackets. (In these cases, if you wish to supply text in place of a variable name, you must enclose the text in double quotation marks.)
- Commands and built-in functions such as #IF and #COMPUTE expect expressions as arguments. When you supply a variable name as an argument to one of these commands or built-in functions, you do not need to enclose a variable name in square brackets. For more information about expressions, see Section 3, Expressions.

TACL evaluates the contents of square brackets before other portions of a statement. This example illustrates the effect of square brackets on invocation sequence. #OUTPUT is a built-in function that sends its argument to the TACL OUT file.

```
18> #OUTPUT #PMSEARCHLIST
#PMSEARCHLIST
19> #OUTPUT [#PMSEARCHLIST]
#DEFAULTS $HOME.SUBV $SYSTEM.SYSTEM
20> [#OUTPUT] #PMSEARCHLIST

#PMSEARCHLIST expanded to:

#DEFAULTS $HOME.SUBV $SYSTEM.SYSTEM
```

In the first case, line 18, TACL expands the #OUTPUT function first because it encounters it first; there are no square brackets in the command line. The #OUTPUT function uses the simple text “#PMSEARCHLIST” as its argument.

In the second case, line 19, TACL expands #PMSEARCHLIST first because of the brackets. The #PMSEARCHLIST built-in variable returns the list of subvolumes in the program and macro search list; TACL then expands #OUTPUT with that text as its argument.

In the third case, line 20, TACL expands #OUTPUT entirely within the limits of the square brackets, producing a blank line of output; then TACL invokes #PMSEARCHLIST.

If you need to output a square bracket from a TACL program, you can use the ~[combination described in Table 2-1 on page 2-3.

Tilde (~)

The tilde is used in combination with another character:

- ~; simulates an end-of-line character; this character allows you to enter multiple TACL commands, separated by ~; combinations, in one line of terminal input.
- ~_ represents a space character when #OUTFORMAT is set to PRETTY. This metacharacter is useful for managing the spacing and alignment of output. For more information, see the #OUTFORMAT Built-In Variable on page 9-274.
- TACL replaces the combination of a tilde and a metacharacter with the metacharacter itself. For example, ~[becomes [, and ~== becomes ==.
- TACL replaces the combination of a tilde and a character that is not a metacharacter with a single question mark. For example, ~A becomes ?

Separator Characters

Most data supplied to TACL must end with a separator character. For example, keywords, file names, and variable names must end with a standard TACL separator; numbers and tokens, on the other hand, need not end with a separator character.

Note. You cannot use these characters in the KEYWORD and TOKEN definitions of the #ARGUMENT built-in function.

Table 2-2, lists the standard TACL separator characters.

Table 2-2. Separator Characters

Character	Name
	Space
,	Comma
(	Left parenthesis
)	Right parenthesis
/	Slash
;	Semicolon
carriage return	Physical end of line
~;	Logical end of line; allows multiple statements per line.

Question Mark (?)

TACL uses the question mark character for two purposes:

- Displaying previous command lines
- Specifying the start of a TACL directive

To start a line with a question mark for any other purpose, use two question marks. TACL then discards the first question mark and any adjacent spaces and treats the remainder of the line as text.

Ampersand (&)

An ampersand (&) at the end of a line of TACL commands or function calls signals that the line continues on the next physical line. This continuation applies to executable statements and comments, with two exceptions:

- TACL does not interpret data transmitted by a process or file opened using #REQUESTER or #SERVER; any special characters are treated as text. If you receive a line that contains an ampersand, TACL does not view the ampersand as a continuation character.
- You cannot use the continuation character with a TACL directive (a line beginning with a question mark).

Note. TACL strips off comments when converting text from external format to internal format, so any ampersand that appears before a comment appears to be at the end of the line. This behavior can cause TACL to treat the next line as a continuation of the previous line.

The maximum line length is 239 characters; if a line is longer than 239 characters, TACL truncates the line to 239 characters.

Template Characters

Templates are special constructs that allow you to perform simple pattern-matching operations. For example, the command

```
FILENAMES c?t
```

displays all three-letter file names in the default volume and subvolume that begin with C and end with T, such as CAT or CUT. The command

```
FILENAMES cat*
```

displays all file names that begin with CAT, regardless of name length, such as CAT, CATALOG, or CATERERS. You can use templates to specify several entities with similar names in a single request. TACL supports templates for file names and DEFINEs.

Subvolume Templates

Several TACL commands and built-in functions allow you to supply subvolume templates in place of actual subvolume names. The TACL command or built-in function then operates on one or more similarly named files. A subvolume specification includes these fields:

```
[[[\node-name.]$volume-name.] subvolume-name.]
```

A subvolume template follows the format of a file name, but contains one or more of the template characters listed in Table 2-3, in place of specific characters in the file name. You can include these template characters in any field of the subvolume specification except the *node-name* field, which does not allow the * character. In addition, template characters cannot match a volume identifier ($) or a field separator (.) in a file name.

File-Name Templates

Several TACL commands and built-in functions allow you to supply file-name templates in place of actual file names. The TACL command or built-in function then operates on one or more similarly named files. A file-name specification includes these fields:

```
[[[\node-name.]$volume-name.] subvolume-name.] file-name
```

A file-name template follows the format of a file name, but contains one or more of the template characters listed in Table 2-3, in place of specific characters in the file name. You can include these template characters in any field of the file-name specification except the *node-name* field, which does not allow the * character. In addition, template characters cannot match a volume identifier ($) or a field separator (.) in a file name.

Table 2-3. Template Characters

Character	Description
?	Matches any single character
*	Matches any number of characters, including none

DEFINE Templates

Most TACL functions that allow DEFINEs allow you to specify templates for DEFINE names. These templates use the same characters as file-name templates, with the added convention that you can use these patterns to refer to all existing DEFINEs:

```
=* or **
```

For more information about DEFINEs, see Section 5, Statements and Programs and the *Guardian User's Guide*.

Operators

Operators perform mathematical or logical operations on values. The operators provided by TACL can be divided into three categories:

- Arithmetic
- Logical
- String

For a complete list of operators, see Section 3, Expressions.

Constants

A constant is a value that you can store in a variable, use as a literal, or specify as part of an expression. There are two types of TACL constants: text constants and string constants. An integer is a special type of text constant.

Text Constants

A text constant is any sequence of these characters:

- Nonmetacharacters from the ISO character set
- Tilde-metacharacter combinations

TACL ignores leading and trailing spaces when interpreting a text constant.

Use a text constant with functions that accept text arguments. For example, the #REPLY built-in function accepts a text argument. Built-in functions that accept text arguments use all the remaining text on the line, with leading and trailing spaces and end-of-line characters removed.

These text constants are valid:

- DATA1
- 456
- Please type a number

An integer is a special type of text constant that consists of a sequence of digits. The sequence can include a prefix that specifies a positive or negative sign. Many integers in this manual are listed with commas for clarification. Do not, however, include commas when you specify an integer. TACL does not accept commas in integers.

Integer-constant:

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

An integer constant can be written as one or more decimal digits (0-9) with an optional sign prefix (+ or -). A TACL integer is similar to the FIXED numeric constant available in

the Transaction Application Language (TAL). The internal representation is a 64-bit fixed-point binary number. The valid range is:

```
-9223372036854775808 to +9223372036854775807
```

This range is called max-int in command descriptions and function descriptions in this manual.

These integers are valid

- 145
- 4
- -89776

TACL uses integer values for logical operations and comparisons. TACL interprets zero as FALSE and nonzero integers as TRUE.

String Constants

A string constant is any sequence of these characters, enclosed within a pair of quotation marks (“ ”):

- Nonmetacharacters from the ISO character set
- Tilde-metacharacter combinations

You can include leading and trailing spaces within the quotation marks. Use a string constant with functions that accept string arguments. In addition, you can use a string constant in place of a variable level in many built-in functions, such as built-in functions that manipulate variables (functions with names that end in “V”). For example, the #LINEFINDV function accepts a string constant.

These string constants are valid:

- “ Error on input ”
- “456”

Reserved Words

A reserved word is a predefined, symbolic name that has special meaning to TACL. Reserved words cannot be redefined; if you try to redefine one, TACL returns an error. Reserved words include:

- All built-in variable names
- All built-in function names

To define a new level of a built-in variable, use the PUSH command or the #PUSH built-in function. You can then specify a new value for the built-in variable while

preserving the former value. For more information about variables and variable levels, see Section 4, Variables.

Note. TACL does not prevent you from redefining TACL commands. You can, for example, define a macro or function with the name TIME-a standard TACL command. You can also load a segment file that has the same name as a TACL command. If you make such a change, TACL will not act in its standard manner; it will, instead, execute your code.

Comments

TACL supports three forms of comments:

- The “double equal” metacharacter: == comment text

  This form causes TACL to ignore all subsequent text until the end of the line, including braces and square brackets but excluding ampersands. To continue the comment on the next line, use the line continuation character (&).

```
== this is a valid comment
== and this == is }{ a valid ][ comment
== this is a valid comment that &
is carried over to the next line }
```

- Pairs of braces: { comment text }

  When TACL encounters an opening brace, it ignores all subsequent text, including double equal signs (==) and square brackets, until it encounters a closing brace. The closing brace must appear on the same line unless you continue the line with the line continuation character (&). An opening brace appearing within the comment text is considered an error.

```
#OUTPUT This is not{this is a valid comment} a comment
{ this is a valid comment that &
is carried over to the next line }
{ this is an invalid comment; it exceeds one
line but has no "&" continuation character }
{ this is an invalid comment; it has two opening {'s }
```

- The COMMENT command: COMMENT *comment text*

  COMMENT is a TACL command that simply ignores its argument unless the argument contains square brackets or braces. COMMENT is provided for compatibility with earlier command interpreters; for new work, the use of braces or double equal signs is recommended because of these special considerations:

  - If the argument contains a square-bracketed function, TACL executes the function, but ignores its result along with the rest of the argument. A single left bracket in the argument is not actually invalid, but causes unpredictable results: TACL assumes it is the beginning of a function invocation and

processes as such all subsequent lines of text until it encounters a right bracket.

- If the text contains braces, the rules for that form of comment also apply.

Comment text is terminated by the end of the command line unless you continue the line with the line continuation character (&):

```
COMMENT This is == a valid }{ comment
COMMENT This is also a valid comment &
that is carried over to the next line
COMMENT {This {thing} is an invalid comment}
```

In all forms of comments, comment text is optional. TACL does not support nested comments.

3 Expressions

An expression can be a text, string, or integer constant, a variable, or a value obtained by combining constants, variables, and other expressions with operators. This section describes operators and how to use them in expressions.

Operators

Operators perform operations on values. The operators provided by TACL can be divided into four categories:

- Arithmetic
- Relational
- Logical
- String

Operators have an order of precedence; when several operations take place within the same program statement, operations with higher precedence are performed before operators with lower precedence. Table 3-1 lists the category, function, and relative precedence of each operator; a lower number indicates higher precedence. To change the order of precedence, use parentheses. A subexpression in parentheses is evaluated before the rest of the expression that contains it.

Table 3-1. TACL Operators (page 1 of 2)

Operator	Function	Type of Operation	Precedence
NOT	NOT	Logical	0
*	Multiplication	Arithmetic	1
/	Division	Arithmetic	1
+	Addition	Arithmetic	2
-	Subtraction	Arithmetic	2
>	Greater than	Relational (integers)	3
<	Less than	Relational (integers)	3
>=	Greater than or equal to	Relational (integers)	3
<=	Less than or equal to	Relational (integers)	3
=	Equal to	Relational (integers)	3
<>	Not equal to	Relational (integers)	3
‘+’	Concatenation	String	3
'>' or '!>'	Greater than	Relational (strings)	3
'<' or '!<'	Less than	Relational (strings)	3
'>=' or '!>='	Greater than or equal to	Relational (strings)	3
'<=' or '!<='	Less than or equal to	Relational (strings)	3

Table 3-1. TACL Operators (page 2 of 2)

Operator	Function	Type of Operation	Precedence
'=' or '!'	Equal to	Relational (strings)	3
'<>' or '!<>'	Not equal to3	Relational (strings)	3
AND	AND	Logical	4
OR	OR	Logical	5

String operators that include an exclamation point (!) are case-sensitive; those without exclamation points make no distinction between uppercase and lowercase letters.

The apostrophes around each string operator are part of each operator and must be present.

Arithmetic Operations

TACL supports arithmetic operations on integers. As with integer-based arithmetic in other languages, perform multiplication operations before division operations to obtain the greatest precision.

Logical Operations

TACL supports logical operations and comparisons for integer and string operands. TACL interprets 0 as FALSE and nonzero integers as TRUE, and typically returns -1 or 1 as the nonzero (TRUE) result for built-in function results. String comparison operators are enclosed in single quotes to differentiate them from numeric operators. For example, assuming STATE is a variable containing text and COUNT is a variable containing a number, this #IF condition could be stated:

```
[#IF STATE '=' "DONE" OR COUNT = 10 |THEN| ...
```

String comparison and numeric comparison yield different results. If, for example, one variable contains 10 and the other contains 010, the two values are equal numerically but unequal in the string sense (they are = but are not '=').

To perform operations interactively, use the COMPUTE command. To obtain the value of an expression for use by other code, use the #COMPUTE built-in function. Note, however, that if a function accepts an expression as an argument, you do not need to specify #COMPUTE. For example, this statement evaluates the embedded arithmetic expression because the #IF function accepts an expression as an argument:

```
[#IF A+1 = 3 |THEN|
...
]
```

To obtain the value of an expression, use the COMPUTE command (interactive) or the #COMPUTE built-in function (programmatic).

If you use a variable as an operand, you do not need to enclose the variable name in square brackets.

Note. TACL evaluates all operands in a logical expression, even after the first FALSE (in an AND expression) or the first TRUE (in an OR expression). Therefore, all variables used in such expressions must first be initialized using #SET or a similar command or built-in function.

---

4 Variables

In many other languages, the term variable refers to a simple variable that contains a single element, such as the number 10, or a more complex variable that contains an set of elements. In TACL, a variable can contain a single element, a set of elements, or information as diverse as a hierarchy of variables or a series of TACL statements.

You can create, set, and delete variables. You can access variables interactively or load them into memory from a file. For more information about loading variables, see Section 5, Statements and Programs.

This subsection provides general information about variables, including a discussion of the stack organization of variables. The remainder of this section provides information about each type of TACL variable.

Along with the variables you create, TACL supplies a set of variables. These variables are called built-in variables and are listed in Section 9, Built-In Functions and Variables.

An Overview of TACL Variables

A TACL variable can contain data, TACL statements, or other information. Table 4-1 lists the seven types of variables. TEXT variables can be used to contain text or procedural constructs.

Table 4-1. TACL Variables and Their Uses (page 1 of 2)

Variable Type	Description
TEXT	Contains text or a sequence of TACL statements. A text variable that contains text is most similar to simple variables used by other programming languages. A TEXT variable that contains TACL statements does not accept arguments, but can be used in a similar way as a MACRO variable (described below) if arguments are not necessary.
ALIAS	Contains the name of a TACL variable. An ALIAS variable allows you to invoke a variable by a different name: the alias name. Any arguments supplied when invoking an alias variable are passed to the referenced function.
MACRO	Contains a sequence of TACL statements. Arguments presented to the macro are substituted, without interpretation or checking, for dummy arguments in the macro text; each argument is referenced by position.
ROUTINE	Contains a sequence of TACL statements. A routine can parse its own arguments (TACL supports several predefined argument types) and can compute its own result.

Table 4-1. TACL Variables and Their Uses (page 2 of 2)

Variable Type	Description
STRUCT	Contains binary data and a set of type definitions that control the conversion of the binary information to and from associated textual representations
DELTA	Contains a sequence of commands for the TACL low-level text manipulator (#DELTA) You cannot execute a DELTA variable as a TACL function.
DIRECTORY	Contains an entire set of variables A DIRECTORY has a hierarchical structure. If used as a function, TACL executes the top level of a variable named EXECUTE.

Variable Names

A variable name can contain from 1 to 31 characters and must start with a letter. The name can contain letters, numbers, underscore characters (_), and circumflexes (^). Variable names are not case-sensitive. These are valid variable names:

```
var1
ems^text^info
write_data
```

Coexisting With TACL Programs Supplied in a TACL Software RVU

To avoid conflicts with TACL software provided as part of a software RVU, adhere to these rules:

- Never create variables whose names begin with a circumflex (^) and never use, in any way, such variables supplied as part of a TACL software RVU.
- Do not create or use variables whose names begin with an underscore (_), except where specifically permitted as a feature of a TACL software RVU.
- Do not create any variables under :UTILS. Similarly, the associated source file, TACLSEGF, can contain only TACL programs supplied as part of a TACL software RVU.
- Do not create any variables under :UTILS_GLOBALS except where specifically permitted.
- Do not push or pop :UTILS or :UTILS_GLOBALS.
- If you modify the use list, ensure that your use list always includes certain directories necessary for the correct operation of TACL programs supplied as part of a TACL software RVU. The USE command automatically does this for you. The

list of necessary directories depends on the TACL software RVU and must not be hard coded in your TACL programs.

Variable Levels

A variable level is an important organizational concept associated with TACL variables. Whenever you create a variable, TACL creates a stack for the variable. Each element of the stack is known as a variable level. The stack organization allows you to create local copies of a variable. You can delete the local copies when you exit the local environment, restoring the variable to its original value.

When you add a new variable level, TACL pushes the stack down by one level; when you remove a level, TACL pops the stack by one level.

Note. Descriptions in this manual sometimes use the term “variable” to mean “variable level,” for brevity.

This diagram illustrates a sample variable, var1, that contains three levels:

Variable Name and Level	Contents
var1.3 (top), also accessible as var1	XYZ
var1.2	A
var1.1	35

Declaring a Variable

Table 4-2 lists TACL functions that allocate and define variables.

Table 4-2. Functions and Commands That Allocate and Define Variables

Command (Interactive)	Function (Programmatic)	Description
PUSH	#PUSH	Adds a level to the top of a variable. If the variable does not exist, PUSH and #PUSH reserve the name; when you use SET VARIABLE or #SET to assign a value to the variable, TACL reserves space for the variable.
-	#DEF	Adds a level to the top of a variable. If the variable does not exist, TACL creates it. You can use #DEF to assign initial values to the variable. To create a STRUCT variable, you must use #DEF.
SET VARIABLE	#SET, #SETV, #SETMANY	Assigns a value to a variable. If this is a new variable, this step actually creates the variable.

In addition, the ?SECTION directive allows you to declare any type of TACL variable. To access such variables, you must LOAD the file that contains them.

The choice of a declaration mechanism depends on whether you want to define the variable interactively, within a file, or in a library. For more information about files and libraries, see Section 5, Statements and Programs.

When you create a variable, TACL creates level 1 of the variable. To create additional levels, push or define the same variable. TACL then creates a new level of the variable and increments the level number.

You can use the #FRAME built-in function at the start of a macro or routine to define a local environment for variables. If you create variables within a frame, you can remove all the variables and their levels by deleting the frame with an #UNFRAME or #RESET FRAMES operation.

A variable remains in existence unless you delete it with a #POP function call, POP command, an #UNFRAME operation, or a #RESET FRAMES operation.

Specifying a Level of a Variable

To specify a level of a variable, append a period and a level number to its name. If you omit the level number, TACL accesses the top level.

You can specify a level number in one of two ways:

- Numbers greater than zero refer to levels relative to the bottom of the stack; they indicate the order of creation of the levels. As an example, name.1 is the bottom level, name.2 is the next to the bottom level, and so on.
- Numbers less than or equal to zero refer to levels relative to the top of the stack; they indicate how far down a level is from the top of the stack. Therefore, name.0 is the top level, name.-1 is the next to the top level, and so on.

This diagram shows the two ways to address levels of the sample variable var1:

Bottom-Up Addressing		**Top-Down Addressing**
var1.3 (top), also accessible as var1	XYZ	var1.0
var1.2	A	var1.-1
var1.1	35	var1.-2

Different levels can contain different types of data. For example, the first level of a variable could contain an alias, the second level could contain text, and the third level could contain a macro.

Deleting a Variable

Table 4-3 lists TACL functions that delete variables.

Table 4-3. Functions and Commands That Delete Variables

Command (Interactive)	Function (Programmatic)	Description
POP	#POP	Removes a level from the top of a variable
KEEP	#KEEP	Removes one or more levels from the bottom of a variable
-	#UNFRAME	Pops all variable levels pushed since the last #FRAME operation
-	#RESET FRAMES	Performs an #UNFRAME operation for all frames with frame numbers higher than they were when a routine was entered If not called from a routine, TACL performs an #UNFRAME operation for all frames with numbers higher than they were when the last prompt was issued.

Accessing Variable Contents

The mechanism for accessing variable contents depends on the context:

- To access the contents of a variable from a location that expects text, enclose the variable name in square brackets ([]). When TACL encounters a variable name in square brackets, it replaces the variable name with the contents of the variable; this action is known as expanding a variable. For example:

  ```
  #OUTPUT [var1]
  ```

  If the variable is a routine that contains TACL statements, TACL replaces the variable name with its associated statements, interprets the statements, and produces results as directed by the statements. The result ends up in place of the variable name, making the result very accessible to your program.

  If you include a level number within the square brackets, TACL expands the contents of the variable level. If you omit the level number, TACL expands the top level of the variable.

- If you are calling a variable as a simple function, then you do not need to surround the function call with brackets. This example shows a #SET function call without square brackets surrounding it:

  ```
  #SET temp 123
  ```

- If a statement within a variable contains an enclosure, include brackets to delimit the enclosure:

  ```
  [#IF level > limit |THEN|
       #OUTPUT Over limit
  ```

```
|ELSE|
     #OUTPUT Value is OK

]
```

- To expand a result or the contents of a variable, include an extra set of brackets beyond any others required by the foregoing rules.

Using a Variable as an Argument

To specify a variable as an argument to a function that expects a variable name, an expression, or a string (passing the variable by name), enter the variable name:

```
#OUTPUTV var1
```

TEXT Variables

A text variable can contain:

- Integer or text data as defined in Section 2, Lexical Elements.
- TACL statements as defined in Section 5, Statements and Programs.

Sample Declarations

This TACL statement creates a TEXT variable called *x*:

```
#PUSH x
```

This TACL statement creates a TEXT variable called *var1* and assigns the value 3 to *var1*:

```
[#DEF var1 TEXT |BODY| 3]
```

This example creates a TEXT variable called *printit*; this variable contains TACL statements:

```
?SECTION printit TEXT
#OUTPUT Here is line one
#OUTPUT Here is line two
#OUTPUT I'm finished!
```

To use this code, you must first load the file that contains the code. For more information, see Section 6, The TACL Environment.

ALIAS Variables

An alias is a limited variable; its contents must be exactly one word-the name of a variable or a file. (Comments of the == and { } forms are allowed, because TACL deletes them; but COMMENT lines are not allowed, because they remain.) An alias is simply a name for something else; for example, the variable S could be an alias for STATUS.

For example, if you replace the TACL STATUS command with a macro called STATUS, you can no longer use the TACL STATUS command. To access the TACL STATUS command, you must make the variable S an alias for :UTILS:TACL:STATUS.

You can also use alias variables to define function keys. For more information, see the *Guardian User's Guide*.

Sample Declarations

This definition defines the variable S as an alias for the STATUS command:

```
[#DEF s ALIAS |BODY| STATUS]
```

After loading this definition, type the letter *s* to obtain status information.

Limitations

An alias must define something TACL can actually execute. TACL generates an error if an alias returns simple text. A macro (described in MACRO Variables) does not have this restriction. For example, given these definitions:

```
34> [#DEF a ALIAS |BODY| Huzzah]
35> [#DEF m MACRO |BODY| Huzzah]
```

you could do this:

```
36> #OUTPUT [m]
Huzzah
```

but not this:

```
37> #OUTPUT [a]
Huzzah
 ^
Expecting ...
```

followed by lists of what the alias could have been, followed by

```
#OUTPUT [a]
          ^
*ERROR* Cannot resolve alias
38>
```

MACRO Variables

A macro variable contains TACL statements; a macro can include conditional logic and can invoke other macros and functions. When TACL encounters a macro name, it replaces that name with the entire contents of the macro and interprets it, performing the specified work.

If you declare variables within the macro and use #FRAME and #UNFRAME to precede and follow their declaration and use, TACL deletes the variables before you exit the macro.

For information about TACL statements, see Section 5, Statements and Programs. For additional examples of TACL macros, see the *TACL Programming Guide*.

Macro Arguments

A macro accesses arguments by position. To access an argument, specify the position, enclosed in percent signs (% n%). For example, %1% references the first argument supplied when the variable is invoked. Table 4-4 lists several ways to access macro arguments from within a macro.

Table 4-4. Macro Arguments

Argument Form	Description
%0%	The name of the macro itself
%n%	The nth argument supplied by the user
%n1 TO n2%	The range of arguments from n1 to n2
%n1 to *%	The range of arguments from n1 up to and including the last argument
%n1 TO n2 BY n3%	The range of arguments from n1 up to and including n2, using the increment n3 to step through the specified portion of the list (For example, %1 TO * BY 2% selects all odd-numbered arguments.)
%n1 TO * BY n3%	The range of arguments from n1 up to and including the last argument, using the increment n3 to step through the specified portion of the list
%*%	All arguments

When you invoke a macro, you supply arguments as a space-separated list. TACL substitutes the real arguments in place of the dummy arguments. If you supply more arguments than are used in the macro, TACL ignores the extra arguments.

Because of the use of the percent sign to denote dummy arguments, if you want to include a literal percent sign in a macro, enter it twice (%%).

Sample Declarations

This code defines a macro called `fn` that calls FILENAMES with the first argument supplied to `fn`:

```
?SECTION fn MACRO
FILENAMES %1%
```

If the argument is not a file-name template, TACL returns an error. You could also use #DEF to define this macro:

```
[#DEF fn MACRO |BODY| FILENAMES %1%]
```

This macro displays each of its arguments on a separate line:

```
?TACL MACRO
#OUTPUT %1% == Output the current argument
[#IF NOT [#EMPTY %2 TO *%] == Test for additional arguments
     |THEN| %0% %2 TO *%   == Call self with first arg. gone
]
```

ROUTINE Variables

A routine variable contains TACL statements. When TACL encounters a routine, it executes the routine and replaces the invocation with the results returned by the routine.

TACL routines support these capabilities that are not provided by text variables or macro variables:

- Sophisticated argument handling capabilities (provided by the #ARGUMENT, #MORE, and #REST built-in functions)
- Generation of the routine's own expansions (provided by the #RESULT built-in function)
- Multiple exit points (provided by the #RETURN built-in function)
- Ability to handle exceptions (provided by the #EXCEPTION, #FILTER, #RAISE, and #ERRORTEXT built-in functions)

TACL executes a routine within a separate buffer; the only result is generated by one or more #RESULT built-in functions within the routine. The important distinction between macros and routines is that a macro invocation returns the entire text of the macro to be executed; a routine invocation returns only what the #RESULT function provides.

You can perform exception handling in routines to retain control in case of error instead of giving up control to the TACL automatic exception-handling logic. For more information, see the *TACL Programming Guide*.

For more information about TACL statements, see Section 5, Statements and Programs. For additional examples of routines, see the *TACL Programming Guide*.

Routine Arguments

A routine does not use dummy arguments (% n%) for its method of argument handling. Instead, use the #ARGUMENT built-in function within the routine to examine and validate arguments and assign their values to variables for use within the function. #ARGUMENT checks arguments one at a time to see if they satisfy any one of a list of argument types. For some types of arguments, such as disk files, ensure that the argument refers to an existing object or that the argument represents a syntactically correct name for an object. (For example, the file associated with a file name must

exist to be a valid file-name argument, but if you specify SYNTAX, TACL checks only for correctness of the file-name syntax.)

If the argument matches a listed type, the argument is placed into a specified variable, and TACL returns a number that indicates the argument type. If the argument does not fit any of the specified types, the routine terminates and generates an error message, listing the types of arguments that were expected.

To parse arguments to a routine, use the #ARGUMENT built-in function. The #ARGUMENT function allows you to define the types of arguments that can be processed by the routine. TACL searches this list from left to right when it processes each argument and returns the position (in the list) of the first argument type that matches the argument.

If you declare variables within the routine, surround their declaration and use with #FRAME and #UNFRAME calls, TACL deletes the variables before you exit the routine.

Sample Declaration

This code defines a routine variable called day_of_the_week that contains a series of TACL statements. The routine accepts a timestamp as its argument; the argument is optional:

```
?SECTION day_of_the_week ROUTINE
#FRAME
#PUSH days timestamp rslt
== Accept one argument that is a timestamp.
#SET rslt [#ARGUMENT /VALUE timestamp/ NUMBER END]
[#CASE [rslt]
|1| SINK [#ARGUMENT END] == A valid timestamp was supplied
|2| == No timestamp; use the current timestamp:
#SET timestamp [#TIMESTAMP]
|OTHERWISE|
== Unexpected value from #ARGUMENT
#OUTPUT Invalid argument
#UNFRAME
#RETURN
]

== Calculate the numbers of days since 0 and since
== the beginning of the week
#SET days [#COMPUTE [timestamp]/(24*60*60*100) ]
== Return the day of the week.
#SET days [#COMPUTE days - ((days/7) * 7)]
#RESULT [#CASE [days]
|0| Tuesday
|1| Wednesday
|2| Thursday
|3| Friday
|4| Saturday
|5| Sunday
|6| Monday
```

```
]
#UNFRAME
```

The SINK commands discard the results of #ARGUMENT and prevent TACL from echoing to the TACL OUT file. A successful evaluation returns 1; otherwise, the routine terminates with an error. You could also use a #SET call to evaluate the results.

Comparing Argument Handling in Macros and Routines

These examples show the difference in argument processing between macros and routines.

This is a sample macro:

```
?TACL MACRO
== This macro does not check syntax or existence of
== file^name.
#FRAME
#PUSH file^name
#SET file^name %1%
#OUTPUT File name is [file^name]
#UNFRAME
```

The macro processes whatever it is given (even if no argument is supplied) and outputs what it is given. Separate coding is required to validate arguments. If you store this macro in a file named HEYMAC, TACL produces this when you run HEYMAC:

```
39 HEYMAC thisfile
File name is thisfile
```

This is a sample routine that performs a similar function:

```
?TACL ROUTINE
== This routine returns an error if there is no argument,
== if the argument does not have correct syntax, or
== if the named file does not exist.
#FRAME
#PUSH file^name
SINK [#ARGUMENT /VALUE file^name/ FILENAME]
#OUTPUT File name is [file^name]
#UNFRAME
```

If you store the routine in a file named ROOTN, TACL produces this when you run ROOTN:

```
40 ROOTN thatfile
File name is \NODE.$VOL.SUBVOL.THATFILE
```

The #ARGUMENT function in the routine requires a file name, so TACL checks that:

- An argument is present in the function call
- The argument has proper syntax for a file name
- A file with the specified name actually exists

The VALUE option of the #ARGUMENT function returns the fully qualified form of the file-name argument.

STRUCT Variables

A STRUCT variable, or structure, is a special-purpose variable that contains a set of named and typed data items that you can access individually or as a group. Use STRUCTs to access the Subsystem Programmatic Interface (SPI), the Event Management Service (EMS), or if you need to store binary data such as 6530 terminal escape sequences.

TACL STRUCT variables support a wide range of data types. STRUCT variables can contain simple data items, arrays, and other structures (called substructures).

Structures usually contain related data items. For example, a structure might contain a process name, the primary CPU,PIN of the process, and its backup CPU,PIN.

Elements of STRUCT Variables

STRUCT variables have these elements:

- A name
- A body that can contain:
  - Single-value data items
  - Arrays
  - Substructures

To declare a STRUCT, use #DEF or ?SECTION to declare a structure body and all items and substructures associated with the body. For each STRUCT, TACL stores two types of information: access information and the actual data. (If you declare a structure using LIKE, the STRUCT contains a pointer to a similar STRUCT that contains access information.) You store and retrieve STRUCT data as text; TACL stores the data in an internal format and performs translation to and from the internal format as needed. You can redefine a STRUCT and specify new access information for your data.

A structure in TACL can contain up to 5000 bytes of data.

Limitations on the Use of STRUCT Variables

Not all TACL commands and built-in functions accept STRUCT variables as arguments. The use of a STRUCT variable is limited to:

- Describing the STRUCT using #VARIABLEINFO (programmatic) or VARINFO (interactive)
- Invoking the STRUCT with the use of square brackets, in which case TACL returns the elements of the STRUCT as a space-separated list

- Assigning values to the STRUCT using #SET (programmatic), SET VARIABLE (interactive), #SETBYTES, or #SETV
- Comparing the STRUCT to another structure or structure item using #COMPAREV (programmatic) or _COMPAREV (interactive)
- Specifying a STRUCT as the source variable for #OUTPUTV (programmatic), OUTVAR (interactive), #APPENDV, or #REPLYV
- Specifying a STRUCT as the destination variable for #EXTRACTV
- Appearing as the prompt or destination variable for #INPUTV
- Initializing STRUCTs using #SSINIT or #SSNULL
- Accepting token entities from #SSGETV
- Supplying token entities to #SSGET(V), #SSPUT(V), or #SSNULL

The use of an item in a structure is limited to:

- Describing the item using #VARIABLEINFO (programmatic) or VARINFO (interactive)
- Invoking the item with the use of square brackets
- Assigning values using #SET (programmatic) or SET VARIABLE (interactive), #SETBYTES, or #SETV
- Comparing the item to another item or structure using #COMPAREV (programmatic) or _COMPAREV (interactive)

Declaring a Structure Body

The syntax for a structure body is either a BEGIN ... END construct or a LIKE construct:

```
{BEGIN declaration [ declaration ... ] END}
{LIKE structure-identifier;}
```

declaration

is the declaration of a STRUCT item: a simple data item declaration, an array data item declaration, a substructure declaration, or a FILLER declaration. These types of declarations are described later.

LIKE *structure-identifier*

specifies that the definition of a structure or substructure is to be identical to that of an existing structure or substructure. *structure-identifier* is the name of an existing variable of type STRUCT.

Considerations

- The BEGIN ... END form can contain declarations for:
  - Simple data items
  - Array data items
  - Substructures
  - FILLER bytes
  - Redefinitions
- The use of LIKE conserves variable space, because like structures and substructures use the same structure-accessing information. Accessing information is not part of a particular structure. The original structure and all like structures merely point to the access information. TACL automatically releases accessing information when it is no longer needed.
- Structure-accessing information must be in the same segment as any structure using it. TACL automatically copies the information to the segment where it is needed, even if it has copied it before; this means that if you have a segment full of definitions to be used by structures in another segment, you should define one structure in that other segment like the original definition, then define any additional structures like that first copy. This precaution ensures that there is only one copy of the structure-accessing information.
- If structure A is defined to be like structure B, changing the definition of B later does not change the definition of A because, although it creates new structure-accessing information for B, A still refers to the original accessing information.
- When manipulating structures in TACL, use data in an appropriate external format. For instance, store and retrieve file names in STRUCTs in external format, although TACL maintains file names in STRUCTs in internal format. In TACL, there is a difference between 12 integers and an internal-format file name; each occupy the same amount of storage, but their representation to a TACL programmer is different.
- TACL aligns structures on word boundaries and allocates storage within a structure:
  - BYTE and CHAR data items in a structure are byte-aligned; all other items are multiple-word items and are word-aligned.
  - A substructure defined by LIKE is word-aligned.
  - A substructure is word-aligned if the first item it contains is word-aligned; otherwise, the substructure is byte-aligned.
  - If a substructure is an array with more than one occurrence, contains a word-aligned item, and would otherwise contain an odd number of bytes, TACL automatically appends one FILLER byte to the substructure to make its length

an even number of bytes. In this case, either all occurrences of the substructure are word-aligned or none are.

Declaring a Simple Data Item

A simple data item declaration associates an identifier with a single-element data item. The syntax for a simple data item declaration is:

```
type identifier [ VALUE initial-value ] ;
```

`type`

is one of these data types:

BOOL	BYTE	CHAR	CRTPID	DEVICE	DEVICE
ENUM	FNAME	FNAME32	INT	INT2	INT4
PHANDLE	SSID	STRUCT	SUBVOL	TRANSID	TSTAMP
UINT	USERNAME	BOOL	UINT		

`BOOL`

is a 16-bit signed value that is either true (represented by -1) or false (represented by 0).

`BYTE`

is an 8-bit unsigned binary integer; its value is in the range 0 to 255.

`CHAR`

is an 8-bit ASCII character.

`CRTPID`

is a 4-word internal-format process ID.

`DEVICE`

is a 4-word internal-format device name.

`ENUM`

is a 16-bit signed, enumerated value whose range is defined by the subsystem and depends on the token number; its value is in the range -32768 to +32767, and its format is the same as for INT type.

`FNAME`

is a 12-word internal-format file name for a disk file, process, or device, in the same form as that produced by the FNAMEEXPAND procedure.

FNAME32

is a 16-word internal file name of the form used by the Distributed Name Service (DNS), consisting of a 4-word internal-format node name followed by a 12-word internal-format local file name.

INT

is a 16-bit signed integer; its value is in the range -32768 to +32767.

INT2

is a 32-bit signed integer; its value is in the range -2147483648 to +2147483647.

INT4

is a 64-bit signed integer; its value is in the range -9223372036854775808 to +9223372036854775807.

PHANDLE

is a 10-integer external representation of a process handle. The ten unsigned integers are separated by periods. Each integer can range from 0 to 65535.

SSID

is a 6-word SPI subsystem identifier; its external representation is one to eight alphanumeric characters or hyphens giving the subsystem owner, followed by a period (.) separator, an integer subsystem number or a subsystem name, a period separator, and an integer version number.

SUBVOL

is the first two parts (8 words) of an internal-format local file name; it can be a disk volume and subvolume, a device name and subdevice name, or a process name and its first qualifier name. The qualifier name can be from one to seven alphanumeric characters, preceded by a number sign. The first character must be alphabetic.

TRANSID

is a 64-bit HP NonStop Transaction Management Facility (TMF) subsystem internal-format transaction ID.

TSTAMP

is a 64-bit, microsecond-resolution Julian timestamp (Greenwich mean time) or an elapsed-time value in microseconds.

UINT

is a 16-bit unsigned integer; its value is in the range 0 to 65535.

USERNAME

is an 8-word internal-format user name in the same form as that produced by the USERIDTOUSERNAME procedure. identifier is a name, 1 to 32 characters in length, which can include alphanumeric, underscore, and circumflex characters; the first character cannot be numeric.

identifier

is a name, 1 to 32 characters in length, which can include alphanumeric, underscore, and circumflex characters; the first character cannot be numeric.

VALUE *initial-value*

specifies the initial value for the field and the value to be given to the field anytime that the STRUCT is cleared. A STRUCT, like any variable, is cleared by setting it to nothing:

```
#SET struct
```

If you omit VALUE, the default initial value depends on the type of the item: Numeric items are set to binary zero; other items are set to ASCII spaces.

For CHAR items, *initial-value* is a character sequence that can appear in either of two formats:

- If it is not preceded by a quotation mark, the character sequence can contain any character except space, semicolon, end-of-line, or any TACL metacharacter.
- If it is preceded by a quote, the character sequence must also be followed by a quote; internal quotes must be doubled (for example, `*Press **return** key*`). The character sequence can contain any character other than a TACL metacharacter.

If you include VALUE for a CHAR item, but supply an incorrect number of initial values, TACL returns a syntax error.

For all other items, *initial-value* is a space-separated list of values appropriate to the type of the item. End-of-line may also be used as a separator. The list ends when the semicolon is reached. If initial-value does not supply enough data for all occurrences of the item, TACL supplies appropriate default initial values for the remaining occurrences.

To be included in a CHAR value, regardless of whether it is enclosed in quotation marks, a TACL metacharacter must be input under PLAIN or QUOTED format, or must be preceded by a tilde: ~[, ~|, ~], ~{, ~}, ~==, or ~~. The tilde is not stored in the structure.

Considerations

These considerations apply to the use of a STRUCT variable to store process information:

- To describe a high-PIN process, use a PHANDLE item in place of a process identifier (CRTPID).
- TACL uses ten unsigned integers, separated by periods, to represent a process handle in external form. Each integer can range from 0 to 65535. Use this external form whenever you send a process handle to TACL (#SSPUT or #SET). This example shows a process handle in TACL external form:

  ```
  1.3.5.7.9.11.13.15.17.19
  ```

- To display a process handle, you can use the OUTVAR command or #OUTPUTV built-in function. In addition, the #VARIABLEINFO built-in function with option TYPE returns type PHANDLE for a process handle field in a STRUCT. You can specify a structure or an array within a structure. For more information, see the next two subsections.

Declaring an Array Data Item

The array data item declaration associates an identifier with a group of data items with the same type. The syntax for the array data item declaration is:

```
type identifier [ ( lower-bound : upper-bound ) ]

[ VALUE initial-value ] ;
```

`type`

is a data type as defined in Declaring a Simple Data Item on page 4-15.

`identifier`

is a name, 1 to 32 characters in length, which can include alphanumeric, underscore, and circumflex characters; the first character cannot be numeric.

`lower-bound`

is a value in the range -32768 to +32767 that defines the number of the first array element; it must be less than or equal to *`upper-bound`*.

`upper-bound`

is a value in the range -32768 to +32767 that defines the number of the last array element; it must be greater than or equal to *`lower-bound`*.

VALUE *`initial-value`*

specifies the initial value for the field and the value to be given to the field anytime the STRUCT is cleared. A STRUCT, like any variable, is cleared when set to a null value.

The initial value used if you omit VALUE depends on the type of the item: Numeric items are set to binary zero; other items are set to ASCII spaces.

For CHAR items, initial-value is the same as that defined for a simple data item (in the previous subsection). If you include VALUE but supply an incorrect number of initial values, TACL returns a syntax error.

For all other items, initial-value is a space-separated list of values appropriate to the type of the item. End-of-line may also be used as a separator. The list ends when the semicolon is reached. If initial-value does not supply enough data for all occurrences of the item, TACL supplies appropriate default initial values for the remaining occurrences.

Consideration

TACL treats a data item declared without bounds as though it had been declared with bounds (0:0, which is one byte long).

Example

This example declares a structure containing various arrays:

```
[#DEF arrays STRUCT
  BEGIN
     INT array^1 (1:100);
     INT array^2 (-5:-1) VALUE 1 2 3 4 5;
     INT array^3 (-90:90);
     CHAR array^4 (0:16) VALUE "This is a ""VALUE""";
     END;
]
```

Declaring a Substructure

A substructure declaration associates an identifier with a structure embedded within another structure or substructure. The syntax for a substructure declaration is:

```
STRUCT identifier [ ( lower-bound : upper-bound ) ] ;
structure-body
```

identifier

is a name, 1 to 32 characters in length, which can include alphanumeric, underscore, and circumflex characters; the first character cannot be numeric.

lower-bound

is a value in the range -32768 to +32767 that defines the number of the first array element of an array data item; it must be less than or equal to *upper-bound*.

upper-bound

is a value in the range -32768 to +32767 that defines the number of the last array element of an array data item; it must be greater than or equal to *lower-bound*.

structure-body

contains the declarations that define the substructure.

Considerations

- You can nest substructures to any practical level. For additional information on substructure alignment, see Declaring a Structure Body on page 4-13.
- TACL treats a data item declared without bounds as though it had been declared with bounds (0:0, which is one byte long).

Declaring FILLER Bytes

A FILLER byte provides a placeholder for structure data or space that your program does not use. TACL aligns some items on word boundaries; you can use FILLER bytes to control the alignment of data.

The syntax for the FILLER declaration is:

```
FILLER num;
```

num

is a number in the range 1 to 5000 that specifies the number of bytes of filler.

FILLER declarations contribute to more readable TACL programs. For example, you can use FILLER bytes:

- To define data that appears in a structure but is not used by your program
- To document word-alignment padding bytes that would be inserted by TACL
- To provide placeholders for unused space

You can access a FILLER item only by accessing a structure containing it; if you do access this structure, TACL treats the FILLER item as type CHAR.

Example

This example shows sample FILLER declarations:

```
[#DEF sample STRUCT
 BEGIN
 CHAR byte (0:2);
 FILLER 1; == Documents word-alignment pad byte
 INT word1;
 INT word2;
 FILLER 30; == Placeholder for unused space
 INT2 integer32;
 END;
]
```

Sample STRUCT Declarations

These examples illustrate several STRUCT declarations.

1. This example declares a two-dimensional array. It consists of two occurrences of a substructure, each of which contains 50 occurrences of a substructure.

```
[#DEF buildings STRUCT
  BEGIN
  STRUCT warehouse (0:1);
    BEGIN
    STRUCT inventory (0:49);
      BEGIN
      INT item^number;
      INT price;
      INT on^hand;
      END;
    END;
  END;
]
```

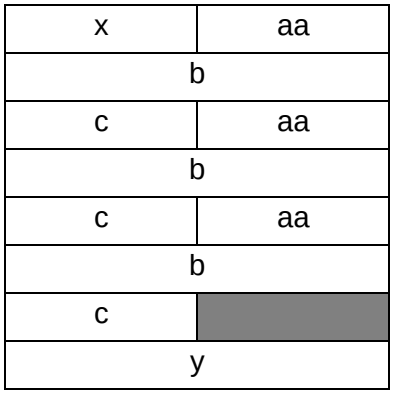
2. This example shows a structure that holds a start-up message.

```
[#DEF startup_message STRUCT
  BEGIN
  INT msgcode;
  STRUCT default;
    BEGIN
    SUBVOL default^name;
    END;
  STRUCT infile;
    BEGIN
    FNAME infile^name;
    END;
  STRUCT outfile;
    BEGIN
    FNAME outfile^name;
    END;
  CHAR param (0:529);
  END;
]
```

3. This example shows storage for substructure occurrences that begin on byte boundaries because the substructure not only follows a CHAR item (“x”) but also starts with a CHAR item (“aa”).

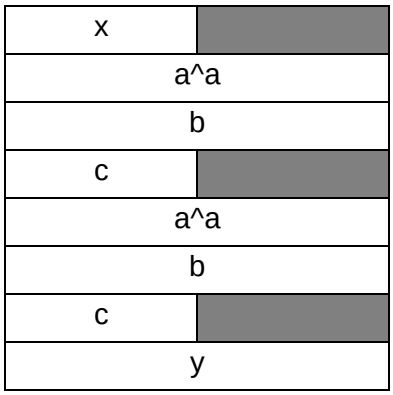
```
[#DEF s STRUCT
  BEGIN
  CHAR x;
  STRUCT sub (0:2);
    BEGIN
    CHAR aa;
    INT b;
    CHAR c;
    END;
  INT y;
```

```
  END;
]
```



4. This example shows storage for substructure occurrences that begin on word boundaries because the substructure starts with an INT item (“a^a”).

```
[#DEF t1 STRUCT
  BEGIN
  CHAR x;
  STRUCT t2 (0:1);
    BEGIN
    INT a^a;
    INT b;
    CHAR c;
    END;
  INT y;
  END;
]
```



5. This example shows substructures defined using LIKE:

```
[#DEF state STRUCT
  BEGIN
  UINT code;
  STRUCT startup_message_in;
    LIKE startup_message;
  STRUCT startup_message_out;
```

```
    LIKE startup_message;
  END;
]
```

Redefining a Structure

A redefinition declares a new name and description for an existing data item or substructure. This functionality is similar to a redefine in TAL or a variant record in Pascal. The syntax for a redefinition declaration is:

```
type identifier [ ( lower-bound : upper-bound ) ]
REDEFINES previous-identifier ;

[ structure-body ]
```

`type`

is one of the data types defined in Declaring a Simple Data Item on page 4-15.

`identifier`

is the name of the new data item that redefines an existing data item, at the same structure-nesting level. The new item can be a simple data item, an array data item, or a substructure.

`lower-bound`

is a value in the range -32768 to +32767 that defines the number of the first array element of an array data item; it must be less than or equal to *`upper-bound`*.

`upper-bound`

is a value in the range -32768 to +32767 that defines the number of the last array element of an array data item; it must be greater than or equal to *`lower-bound`*.

`previous-identifier`

is the name of a data item previously declared at the same structure nesting level. You cannot specify array bounds with this name.

`structure-body`

is used only when redefining a substructure (`type` is STRUCT); it contains the declarations that describe the new substructure.

Considerations

- A redefinition always starts at element zero of the previous item regardless of the bounds of that item; this means that the previous item must at least have an element 0.
- The data area of a redefinition must be exactly the same as, or entirely within, the data area of the previous item.

- The new item must be capable of the same alignment as the previous item.

△ **Caution.** Data stored by one definition might not be readable when retrieved by using another definition. TACL does not ensure that the data being retrieved is valid under the definition by which it is being retrieved.

Examples

Examples of redefinition declarations follow.

1. This example redefines part of an INT array as an INT2 array. The redefinition begins at a(0):

```
[#DEF s STRUCT
  BEGIN
  INT a(-2:3);
  INT2 b(1:2) REDEFINES a;
  END;
]
```

Definition	Redefinition
a(-2)	
a(-1)	
a(0)	b(1)
a(1)	
a(2)	b(2)
a(3)	

2. This example shows a substructure redefinition; the new substructure is smaller than the previous substructure:

```
[#DEF str1 STRUCT
  BEGIN
  STRUCT sub1;
    BEGIN
    INT int1;
    END;
  STRUCT sub2 REDEFINES sub1;
    BEGIN
    CHAR chr1;
    END;
  END;
]
```

Definition	Redefinition	
int1	chr1	

3. In this substructure redefinition, both substructures (“b” and “c”) have the same alignment as required. In this case, both begin on an odd-byte boundary:

```
[#DEF a STRUCT
  BEGIN
  CHAR xx;
  STRUCT b;
    BEGIN
    CHAR yy;
    END;
  STRUCT c REDEFINES b;
    BEGIN
    CHAR zz;
    END;
  END;
]
```

4. This example redefines the format of a substructure record:

```
[#DEF name^record STRUCT
  BEGIN
  STRUCT whole^name;
    BEGIN
    CHAR first^name (0:10);
    CHAR middle^name (0:10);
    CHAR last^name (0:15);
    END;
  STRUCT initials REDEFINES whole^name;
    BEGIN
    CHAR first^initial;
    FILLER 10;
    CHAR middle^initial;
    FILLER 10;
    CHAR last^initial;
    FILLER 15;
    END;
  END;
]
```

Setting or Altering Structured Data

To initialize data in a STRUCT, use #DEF or #SET. To change values in the STRUCT, use #SET.

When storing data in a structure item, you must enter it in a representation appropriate for the type of that item:

- If you are storing data in a substructure or array (except of type CHAR) you must enter it into a space-separated list of representations appropriate for the types of those items. If the data is shorter than the substructure or array, TACL sets remaining numeric binary items to binary zero and sets other types of items to spaces.
- Data that is being stored into an array of type CHAR is not separated by spaces and may even include spaces. If the data is exhausted, TACL sets the remainder of the array to spaces.

These examples are based on this structure:

```
[#DEF tacl^files STRUCT
  BEGIN
  STRUCT file (1:3);
    BEGIN
    FNAME name;
    UINT code;
    END;
  END;
]
```

Use the #SET built-in function to specify values:

```
#SET tacl^files:file(2):name taclbase == set a data item

#SET tacl^files:file(1) tacl 100 == set substructure values

== set structure values
#SET tacl^files tacl 100 taclbase 101 taclinit 101

== set structure values from variables
#SET tacl^files [prg] [prgcd] [lib] [libcd] [mac] [maccd]
```

Accessing Structured Data

To display structured data in a stylized format, use #OUTPUTV. To invoke the STRUCT (expand its contents into a space-separated list), surround the STRUCT name with square brackets.

To access a data item in a structure, specify the fully qualified identifier of the structure item, using this form, with or without array indexes:

struct-name [:*substruct-name* ...] :*item-name*

A structure identifier cannot contain any spaces.

All indexes must be within the range declared for the data item. If you try to use an index outside the range declared for a data item, TACL returns an error.

An example of an identifier of a structure item is:

```
record.-1:table(1):item([x])
```

where .-1 indicates that you are accessing the next-to-the-top level of the variable record; x is a numeric variable containing a valid index.

You can use the #SETV built-in function to copy structures (as well as other types of variables). The output variable level must already exist; its original type and data are lost. The output variable level becomes a structure identical to the input structure and has its own copy of the data.

This is a brief summary of what you can and cannot do when copying STRUCT data.

Suppose you had created this STRUCT to contain an escape control character (ASCII 27) followed by a vertical-line character; this combination clears the screen when sent to 653*x* terminals. To assign the data to a text variable for ease of use, enter:

```
[#DEF makecs STRUCT
  BEGIN
  BYTE b(0:1) VALUE 27 73;
  CHAR c(0:1) REDEFINES b;
  END;
]

#PUSH cs
```

To do so, you could use one of these #SET calls:

```
#SET cs [makecs:c(0:1)]

#SETV cs "[makecs:c(0:1)]"
```

but you could not use this statement:

```
#SETV cs makecs:c(0:1)
```

You can assign structures or substructures with #SETV, but not specific data items of a structure. (If you had defined B and C as substructures of MAKECS, you could use the preceding function call, but it would change CS from a simple text variable to a STRUCT identical to MAKECS). Nor could you use:

```
#SETBYTES cs makecs:c(0:1)
```

because CS is a text variable and #SETBYTES requires that both source and destination be structured variables.

If you had a STRUCT defined as:

```
[#DEF arrays STRUCT
  BEGIN
  INT array^1 (1:100);
  CHAR array^2 (0:16) VALUE "This is a ""VALUE""";
  END;
]
```

you could copy specific elements of array^1 or array^2 to another variable in this way:

```
#SET halfnums [arrays:array^1(1:50)]

#SET firstword [arrays:array^2(0:3)]
```

Data retrieved from a structure item is presented in a standard representation appropriate for the type of that item.

Data retrieved from a substructure is presented in a space-separated list of standard representations appropriate for the types of those items. Spaces are not inserted between CHAR items, however.

△ **Caution.** If one or more of the elementary items consists of or contains spaces, be careful if you later access this data with commands or functions that use spaces as delimiters; otherwise, TACL might overlook an item.

These examples are based on the structure defined in Setting or Altering Structured Data on page 4-25.

To access a data item, surround the structure and item names in square brackets. For example:

```
== Access elements in a structure
[tacl^files:file(2):name] == yields taclbase
[tacl^files:file(1)] == yields tacl 100

== Access the entire structure structure
[tacl^files] == yields tacl 100 taclbase 101 taclinit 101

== Distribute structure items into variables
#SETMANY prg prgcd lib libcd mac maccd , [tacl^files]

== Copy a substructure to a substructure
#SET tacl^files:file(1) [tacl^files:file(2)]
```

△ **Caution.** Use care when moving data between structures and variables of other types unless those variables are used with #SERVER or #REQUESTER or are merely temporary variables for copying the data of one structure to another. TACL interprets data differently for STRUCT variables than for other types of variables. For example, if you process such data with #DELTA while the data is in a variable that is not a structure, #DELTA interprets any bytes holding binary zero as line end characters, and reshapes the data into multiple lines when putting it back into a variable. In short, do not process structure data outside a structure except to perform I/O operations with it.

To fill one STRUCT with data already present in another structure, use #SETBYTES.

You can use the #SETV built-in function to copy structures (as well as other types). The output variable must already exist. After the data has been copied, the original

type and data of the output variable is lost; the variable becomes a structure like the input structure and has its own copy of the data.

DIRECTORY Variables

Directory variables allow you to specify a hierarchical organization for variables that reside in a segment file; directories can contain directories, which in turn can contain other directories. The maximum nesting depth for directories is 16 levels. The root of the tree is the home (:) character. TACL supplies the :UTILS directory, which includes the :UTILS:TACL directory where all TACL variables reside.

Directory variables allow you to keep your variables grouped together in a segment file, for your use or for sharing with other users. Segment files are explained in Section 5, Statements and Programs.

Declaring a Directory Variable

To create a directory, use #DEF or ?SECTION (in a library) and specify the variable type as DIRECTORY. You can also create directories by creating variables. For example:

```
PUSH :a:b:c:d
```

creates these directories, if they do not already exist:

```
:a
:a:b
:a:b:c
```

You cannot, however, create a root (:) directory; do not issue a PUSH : command.

Accessing a Directory Variable

To refer to a specific variable in a directory, you name, in order, all the directories on the direct path from the root to the specific variable. Start the name with a colon at the beginning (which identifies the root directory) and place a colon after each directory name. The name cannot contain spaces. Such a name is called a full path name. For example, the full path name for the RUN command, which is a TACL variable stored in the TACL directory variable in the UTILS directory variable, is :UTILS:TACL:RUN.

As an alternative, you can use a partial path name. A partial path name does not begin with a colon, and it contains only as much of the full path name as is necessary to properly identify the variable. On encountering a partial path name, TACL does one of the following:

- If the partial path name is being supplied to #DEF, #PUSH, PUSH, or ?SECTION (in a library), TACL behaves as though the partial path name were preceded by the directory named in the most recent HOME command. For example,

  ```
  HOME :a:b
  PUSH c
  ```

  is equivalent to

  ```
  PUSH :a:b:c
  ```

  If the partial path name is being supplied to any other command, TACL acts as though the partial path name were preceded by the directory named in the most recent HOME

command, and then successively by the directories named in the most recent USE command, until it finds a variable by that name. For example,

```
HOME :a:one
USE :b :b:two : :c:d:three
OUTVAR d
```

outputs the first one of these that it finds:

```
:a:one:d
:b:d
:b:two:d
:d
:c:d:three:d
```

Directories Supplied With TACL

TACL keeps product variables separate from your user variables by organizing them into directories within the root directory (:). A directory named TACL exists in :UTILS; it contains all variables that define TACL commands. A directory within TACL, called ^UTILS, contains helper variables that are not intended for your direct use. By convention, variables whose names begin with "^" or "_" are reserved for use by a TACL software RVU. You can use variables beginning with "_" in accordance with their definitions, but you should never use a variable beginning with "^" in any way at all. If a directory name begins with "^", you should not use anything in that directory regardless of its name.

The :UTILS directory contains directories for all software products on your system that include TACL programs. The :UTILS_GLOBALS directory is used by software products that include TACL programs and that need to maintain writable global TACL variables.

Note. Do not create, delete, or change any variables anywhere under these two directories. Do not create, delete, or change the directories themselves, unless the documentation for a software product explicitly states that you can. (Some products allow you to modify certain of their variables in :UTILS_GLOBALS.) This means you should not #PUSH :UTILS.

For additional information about directories supplied with the TACL software, see Section 6, The TACL Environment.

DELTA Variables

DELTA variables contain a sequence of commands that are understood by the #DELTA low-level character editor. Most #DELTA capabilities can be performed using string-handling and character-handling commands and functions. For more information about #DELTA, see #DELTA Built-In Function on page 9-111.

5 Statements and Programs

A TACL program consists of one or more statements. A TACL statement consists of one of these:

- A function call
- A directive

This section describes the syntax for TACL statements and directives, and then discusses how to create TACL programs, handle completion codes, and process errors.

For additional examples of TACL programs, see the *TACL Programming Guide*.

Function Calls

This subsection describes each of these types of statements, and includes a list of arguments that are common to built-in TACL functions.

A function call has this syntax:

```
function-call [ argument [ argument ... ] ] [ enclosure ]
[ comment]
function-call
```

is the name of a TACL command or built-in function, or a user-defined TACL program. The syntax for each command is listed in Section 8, UTILS:TACL Commands and Functions. The syntax for each built-in function is listed in Section 9, Built-In Functions and Variables.

You must enclose the function call in square brackets if:

- You want to use the expansion of the function call for further processing.
- The function call includes an enclosure.
- The function call spans more than one line.
- The function call requires brackets to process an argument that references a multiple-line variable. (Some function calls, such as #OUTPUT, expand their arguments and use only the first line unless you surround the function call with square brackets. TACL then tries to execute the second and following lines of the expanded variable.)

If none of the foregoing are true, the function call need not be enclosed in brackets.

argument

is text, a string, a function name, or other construct as defined by the function call. If you supply a function name as an argument, the name must be enclosed in square brackets. For definitions of text and strings, see Section 2, Lexical Elements.

enclosure

specifies that TACL defers expansion of enclosed code until a specific path is selected. (See Considerations.) An enclosure can be used only in a #CASE, #DEF, #IF, or #LOOP built-in function call. An enclosure consists of one or more pairs of labels and text:

label

A label is a text constant or a space-separated list of text constants, enclosed in vertical lines. Each label precedes and identifies a given portion of text (which may be empty).

The text within the vertical lines depends on the type of statement; for example, the label used with the #DEF built-in function is |BODY|.

text

is a string of text that can include one or more function calls as defined previously. TACL expands this text if the associated label is selected by the condition in the controlling #CASE, #DEF, #IF, or #LOOP built-in function. Any square brackets within an enclosure must balance. The last enclosure within the controlling function call is terminated by the right square bracket (]) that ends the function call.

comment

is a valid comment, initiated with double equal sign characters (==) or surrounding braces ({}), which can include text. (The COMMENT command is a function call.)

Considerations

- Note that function-call can be a built-in function or command supplied in the TACL software or a user-defined TACL program; all are handled in the same manner.
- When you enter a program name (such as RELOAD or USERS) as your function call, TACL performs an implicit RUN command. TACL starts the program and then provides the new process with the parameters you entered after the program name. The process then carries out your instructions.
- You can place more than one function call on a physical line. To separate the statements, use a logical end-of-line marker (~;).
- TACL interprets each line or logical line (~;) as the end of a statement unless:
  - There are unbalanced left square brackets, in which case TACL processes lines until all brackets are balanced.
  - The line ends with a continuation character (&), in which case TACL treats the next line as a continuation of the first line.
- For information about how TACL interprets each line, see How TACL Interprets Statements on page 5-11.

- To call your program recursively, use %0% for a macro or #ROUTINENAME to obtain the name of your routine. For examples, see the *TACL Programming Guide*.
- The ?FORMAT directive and #INFORMAT built-in function affect how TACL interprets an argument. Under control of the QUOTED input format, STRING arguments allow inclusion of TACL metacharacters (square brackets, vertical line, braces, and double equal sign) as ordinary data characters; quotation marks do not override metacharacters under control of TACL or PLAIN input format.
- Each logical line in a TACL program can contain a maximum of 239 characters. To extend an 80-character physical line to 239 characters, you can use the continuation character (&), or you can enclose the statement within square brackets ([]). Each physical line length depends on the terminal device. The 65*nn* terminal uses a line length of 80 characters by default. If a function call and its associated arguments exceed the maximum physical line length, you can enclose the statement within square brackets ([]) to treat multiple physical lines as a single statement.
- When TACL encounters an enclosure, it departs from its standard method of evaluating bracketed text. Instead of expanding bracketed text, TACL defers evaluation of all statements in an enclosure until the controlling information is processed. For example, in an #IF statement, TACL evaluates the expression before invoking the code associated with |THEN| or |ELSE|.

Example

This example shows an #IF built-in function call with |THEN| and |ELSE| labels:

```
[ #IF x > 3 |THEN|
#OUTPUT Incorrect number
|ELSE|
#OUTPUT x = [x]
]
```

File-Name Arguments

For file-name arguments, these guidelines apply:

- TACL creates a file name based on what you specify and what the current defaults are. To specify a file that is not in the current default subvolume (or volume), include the appropriate subvolume (or volume) name. If you do not specify the node name, then TACL uses the current default system. For example, if your current defaults are \LOCAL.$WORK.MINE and you want to specify the file TIMING in subvolume IGNITION of volume $TUNEUP on the remote system \AUTOS, you must specify:

  ```
  \AUTOS.$TUNEUP.IGNITION.TIMING
  ```

- TACL performs file-name expansion for partial file names that you specify as arguments in calls to commands and built-in functions that accept file names. A partial file name is one that omits the node name, volume, or subvolume of a full

file name. If, however, you specify a volume name, you must specify a subvolume name. This file name is not syntactically correct:

```
$VOLUME.FILE
```

If you specify a partial file name, TACL expands the file name using the current default names for system, volume, and subvolume where necessary. If you specify MYFILE, for example, TACL assumes you mean the file named MYFILE in the current default subvolume, volume, and system.

For more information about file-name expansion, see the *Guardian User's Guide*.

- Many built-in functions accept file-name templates in place of actual file names, so that the command or function can operate on a number of similarly named files. For information about file-name template characters, see Section 2, Lexical Elements.

Device-Name Arguments

TACL evaluates a device name based on what you specify and what the current defaults are. For file-name arguments, if you do not specify a node name, then TACL uses the current default system.

Process Identifier Arguments

A process identifier identifies a process within a system or in a network. TACL uses these types of process identifiers:

- Process name

  A process name identifies a process or process pair within a node. The name is an alphanumeric string up to five characters long, preceded by a dollar sign, and can be preceded by a node name. The first character must be alphabetic. TACL uses process names in displays and interactive requests. For each named process or process pair, the format is $ *process-name*; for example:

  ```
  $ABC
  \SYSTEM.$PRC12
  ```

  The D-series process name is similar to a C-series process name, but a five-character name is allowed for remote processes. (The syntax for D-series remote process names matches the syntax for local names.)

  TACL evaluates process names based on what you specify and what the current defaults are. If you do not specify a node name, then TACL uses the current default system.

- CPU,PIN (also known as a PID)

  The combination of a CPU number and process identification number (PIN) identify a process on a node. This combination is the only way to identify an unnamed process on C series or earlier software.

- Process handle

A process handle contains ten unsigned integers that identify a single named or unnamed process among all processes that are running or have run in one or more system nodes. The process handle is an internal form of identification for a process. System messages, completion code STRUCTs, and SPI buffers contain process handle information.

Note. The format of a process handle is defined by the TACL software and is subject to change in future RVUs.

TACL creates a process name based on what you specify and what the current defaults are. If you do not specify a node name, then TACL uses the current default system.

DEFINEs as Arguments

A number of TACL functions accept DEFINEs as arguments. A DEFINE is a named set of attributes and associated values. In a DEFINE (as with an ASSIGN command), you can specify information to be communicated to processes you start. The operating system (file system or I/O processes) usually processes DEFINEs, while application programs or run-time libraries process ASSIGNs.

TACL stores DEFINEs in its process file segment (PFS) and can propagate them to processes you start. Some TACL functions permit you to use templates in place of actual DEFINE names. For information about DEFINE templates, see Section 2, Lexical Elements Commands and built-in functions that support DEFINEs are identifiable by the appearance of DEFINE or DEFMODE in their titles. For more information about DEFINEs, see individual command and function descriptions in this manual, the *TACL Programming Guide*, the *Guardian Programmer's Guide*, and the *Guardian User's Guide*.

Directives

TACL directives are for use only in files that contain TACL statements. There are four types of directives:

- ?BLANK
- ?FORMAT
- ?SECTION
- ?TACL

All TACL directives start with a question mark (?).

Note. Question marks are used for two purposes in TACL: for editing previous command lines and for specifying the start of a TACL directive. To start a line with a question mark for another purpose, such as loading DDL commands, use two question marks. TACL discards the first question mark and any adjacent spaces and treats the remainder of the line as text.

?BLANK Directive

Use the ?BLANK directive to insert a blank line into a variable; this can be useful when you load text that is to be displayed. The syntax is:

```
?BLANK
```

?FORMAT Directive

Use the ?FORMAT directive to specify how TACL interprets metacharacters in the TACL statements following the directive. The ?FORMAT directive is similar to the #INFORMAT built-in variable, but acts on statements in a file instead of text from the IN file. The syntax is:

```
?FORMAT { PLAIN | QUOTED | TACL }
```

`PLAIN`

causes TACL to interpret metacharacters and all other characters in the file as ordinary text. For example, braces and double equal signs are read as such and are not interpreted as comments in PLAIN mode.

`QUOTED`

causes TACL to interpret metacharacters as metacharacters (the same as the TACL option), unless text with metacharacters is enclosed in quotation marks. In this case, TACL treats metacharacters as if they were ordinary text (tildes are not needed).

`TACL`

causes TACL to interpret metacharacters as metacharacters and store them in internal notation. TACL reads square brackets as the beginning and ending of an invocation. A vertical line indicates a label in an enclosure. TACL reads braces and double equals as comments.

Using a tilde causes TACL to interpret the next character as plain text, rather than a delimiter; for example, TACL reads ~[as an ordinary open square bracket, rather than the beginning of an invocation. To use a tilde character as text, enter it twice (~~). The tilde has no effect on its own, but only in conjunction with other characters.

When using the TACL format, you can put several commands on a single line by separating the commands with a tilde and a semicolon (~;). TACL translates these metacharacters into internal end-of-line characters.

You can also use a tilde and an underscore (~_) when ?FORMAT is set to TACL. TACL translates this notation into an internal space, which is printed as a space if you use the PRETTY option with #OUTFORMAT; otherwise, the tilde and underscore are treated as ordinary characters.

Considerations

- The ?FORMAT value for a library file is set to TACL at every ?SECTION directive.
- The ?FORMAT value for a macro or routine file that starts with a ?TACL directive is set to TACL at the beginning of the file.

Example

This example shows the effects of the ?FORMAT directive:

```
?SECTION this^section ROUTINE
==
== PLAIN format allows a multiple-line #OUTPUT function to
== output text containing special characters without having
== to use tildes
==
#SET #OUTFORMAT PLAIN == Set display format to PLAIN

[#OUTPUT Using ?FORMAT PLAIN:
?FORMAT PLAIN

+------------------------+
| COMMENT [ <argument> ] |
+------------------------+

?FORMAT TACL
]
#OUTPUT
[#OUTPUT Using ?FORMAT QUOTED:

?FORMAT QUOTED

"+------------------------+"
"| COMMENT [ <argument> ] |"
"+------------------------+"

?FORMAT TACL
]
#OUTPUT
#OUTPUT

#OUTPUT ?FORMAT TACL, with quotes: "[#MYTERM]"
#OUTPUT ?FORMAT TACL, without quotes: [#MYTERM]
#OUTPUT

?FORMAT PLAIN
#OUTPUT ?FORMAT PLAIN, with quotes: "[#MYTERM]"
#OUTPUT ?FORMAT PLAIN, without quotes: [#MYTERM]
#OUTPUT

?FORMAT QUOTED
#OUTPUT ?FORMAT QUOTED, with quotes: "[#MYTERM]"
#OUTPUT ?FORMAT QUOTED, without quotes: [#MYTERM]
#OUTPUT

?SECTION next^section ROUTINE
== Format reverts to TACL because this is a new section
```

```
== This output appears as "$<terminal name>"
#OUTPUT "[#MYTERM]"
```

If you load the preceding file and invoke this section, TACL produces this output when #OUTFORMAT is set to PLAIN:

```
14> this^section

Using ?FORMAT PLAIN:

+------------------------+
| COMMENT [ <argument> ] |
+------------------------+

Using ?FORMAT QUOTED:

"+------------------------+"
"| COMMENT [ <argument> ] |"
"+------------------------+"

?FORMAT TACL, with quotes: "$LM1.#ZWN0009"
?FORMAT TACL, without quotes: $LM1.#ZWN0009

?FORMAT PLAIN, with quotes: "[#MYTERM"]
?FORMAT PLAIN, without quotes: [#MYTERM]

?FORMAT QUOTED, with quotes: "[#MYTERM]"
?FORMAT QUOTED, without quotes: $LM1.#ZWN0009

15>
```

?SECTION Directive

Use the ?SECTION directive to signal the beginning of a variable definition in a file. This directive allows you to create a library that contains definitions for many variables. The syntax of the directive is:

```
?SECTION variable-name variable-type
```

variable-name

specifies the name associated with the following lines of text. For information about valid variable names, see Section 4, Variables.

variable-type

specifies the type of TACL variable: TEXT, ALIAS, MACRO, ROUTINE, STRUCT, DIRECTORY, or DELTA.

Considerations

- To cause TACL to interpret or execute the contents of the file, use the LOAD command or #LOAD built-in function to load the contents into memory. This

example shows how to load two libraries, retaining only one level of each variable in the libraries:

```
17> LOAD /KEEP 1/ mykeys mymacs
```

- To invoke a variable after it is loaded, type the variable name.
- To process a file, TACL pushes the variable for each ?SECTION directive, sets its contents to type, and makes body the new top-level definition of the variable.
- As a library is loaded, comments are removed to conserve variable space. Any line that is blank, or that becomes blank because of comment removal, is discarded.
- For additional information about creating and accessing a file with the ?SECTION directive, see Creating Program Files on page 5-12.

?TACL Directive

Use the ?TACL directive to specify that the TACL statements following the directive in a file are the contents of a TACL variable. The syntax of the directive is:

```
?TACL variable-type
```

variable-type

specifies the type of TACL variable: TEXT, ALIAS, MACRO, ROUTINE, STRUCT, DIRECTORY, or DELTA.

Considerations

- The ?TACL directive, if specified, must be the first line of the file.
- To cause TACL to interpret or execute the contents of the file, type the file name.
- For additional information about creating and accessing a file with the ?TACL directive, see Creating Program Files on page 5-12.

TACL Programs

A program consists of one or more statements. You can enter the statements interactively or store them in a file. This subsection contains information about TACL programs:

- The structure of a TACL program.
- A description of how TACL interprets statements.
- Instructions for creating and accessing three types of program files.
- A description of completion code information.
- A description of TACL error types.

Program Structure

When you write a TACL program, you define the program as a type of TACL variable. There are four types of variables that can contain executable statements: TEXT, MACRO, ROUTINE, and ALIAS. A program can contain of one or more TACL statements:

- A directive, ?TACL or ?SECTION, that indicates that the following lines define a program.
- A #FRAME call that defines a local environment.
- A series of #PUSH or #DEF statements that define data variables and subprograms. (TACL does not require you to place definitions at the start of a program, but placing definitions at the start can increase readability of the program.)
- Additional TACL statements; the main body of the program.
- An #UNFRAME or #RESET FRAMES call if the program called #FRAME earlier.

TACL is an interpretive language; you do not compile or bind TACL programs. TACL programs execute as part of your interactive TACL process unless you direct them to run independently (in the background).

Sample Program

This program, of type macro, purges a file:

```
?TACL MACRO
#FRAME == Establish a local environment for variables
#PUSH err == Declare a variable called "err"

== Set err to the result of the #PURGE function:
#SET err [#PURGE %1%]

== Display the results of the purge operation:
[#IF NOT err |THEN|
  #OUTPUT Purge of file %1% complete!
|ELSE|
  #OUTPUT Error [err]
]
#UNFRAME == Delete the local environment
```

A macro accepts positional arguments; %1% refers to the first argument supplied when you run the program.

To run this program, type the name of the file that contains the TACL statements, followed by the name of an existing file that you want to purge. If, for example, the statements are stored in a file called PRG, you would type this to purge a file called TEMP:

```
18> PRG TEMP
Purge of file TEMP complete!
19>
```

If you try to purge TEMP a second time, TACL returns an error message:

```
19> PRG TEMP
Error 11
20>
```

An error 11 indicates that the file was not found:

```
20> Error 11
011 file not in directory or record not in file, or the
specified tape file is not present on a labeled tape
```

How TACL Interprets Statements

TACL requires read access to interpret a TACL program. TACL uses a 32,767-byte text buffer to execute statements. TACL executes statements in this sequence:

1. If the text buffer is empty, TACL prompts and waits for input. The input is placed into the TACL text buffer, the place where function execution actually takes place. The text buffer may contain from zero to many lines of text. If the text buffer is not empty, TACL examines the contents of the buffer, starting with the first line and moving forward through the buffer.
2. TACL examines the current line of the text buffer from its beginning, searching for square brackets.

   If TACL encounters square brackets, TACL invokes the first element in that line whose closing bracket is found, ignoring any function calls that are inside enclosures. For macro variables, TACL substitutes dummy arguments as necessary. TACL replaces the element with its result (if it was a built-in function or other routine) or its expansion, starts again at the beginning of the line, and repeats the process.

   If the current line of the text buffer does not contain any square brackets, TACL executes the entire line as though the entire line had been surrounded by a pair of square brackets. TACL replaces the function call or line of text with the result of the function. The result can be empty, can contain part of a line of text, or can contain many lines of text.
3. TACL returns to Step 1.

As you can see from Step 2, TACL does not execute function calls within enclosures. Enclosures usually contain TACL statements; functions typically expand enclosures and return unexecuted TACL statements. When TACL returns to Step 1, the code is available for execution. Enclosures are used with a functions that control program flow or define variables; they are not used with any commands.

During execution, the space available in the text buffer diminishes as routines and macros are nested. Also, TACL might need up to half of the available area of the text buffer as a scratch area.

If your code exceeds the size of the text buffer, TACL returns a “Text buffer overflow” error. Common causes are:

- Very large macros or routines. As a rule, macros should never exceed 15,000 bytes after dummy argument substitution; routines should never exceed 15,000 bytes. Nesting reduces these numbers further.
- Uncontrolled nesting of macros. (If, however, you construct a recursive macro so that it invokes itself in its last statement, each instance of the macro vanishes from the text buffer before the next one begins and does not increase the contents of the text buffer.)
- Matching too many file names with the #FILENAMES function. TACL stores the information in the text buffer before displaying it. Loading too much data (such as large subvolumes of file names).
- Using #CHARxxx, #LINExxx, or #DELTA built-in functions on data larger than 15,000 bytes.

Creating Program Files

You can store TACL programs in one of three ways:

- One program in an edit file, executed by referencing the file name. The first line must be a ?TACL directive.
- Several programs in a single edit file, known as a library file, executed by loading the file into variables and then referencing the name of the variable (defined in the code). A library file can also contain other types of variables. Each variable starts with a ?SECTION directive that assigns a name to the variable.
- One or more programs in a segment file, executed by attaching the file (making them accessible to TACL) and then referencing the variable name associated with the program.

You can set up your TACL environment so that it loads variables from edit files or segment files automatically at logon time.

Creating a Single-Variable Program File: The ?TACL Directive

To store a single variable in a file, create an edit-format file whose first line is a ?TACL MACRO or ?TACL ROUTINE directive. To invoke the text, macro, or routine, type the file name.

The name of the subvolume containing the program file must be in #PMSEARCHLIST, or you must qualify it fully, to enable TACL to find it and invoke the file.

When you invoke a ?TACL ROUTINE file, TACL automatically pushes a variable called :_TACL_ROUTINE and reads the file into it; TACL pops :_TACL_ROUTINE when the routine exits. Thus, :_TACL_ROUTINE becomes the name of the active routine.

This feature is useful for recursion: a routine stored in a ?TACL ROUTINE file cannot determine the name of the file it is stored in, but the #ROUTINENAME built-in function,

if executed in a routine stored in a ?TACL ROUTINE file, returns :_TACL_ROUTINE, allowing the routine to invoke itself.

Creating a Library of TACL Statements: The ?SECTION Directive

A library is an edit file that contains a sequential list of one or more TACL variable definitions. For each variable, enter a ?SECTION directive at the beginning of the definition, followed by the name of each variable and its type. (The type is not limited to procedural variables; you can define any type of variable-ALIAS, DELTA, DIRECTORY, MACRO, ROUTINE, STRUCT, or TEXT-in a library file). On subsequent lines, enter the statements associated with the variable.

To load your library into the variables named in ?SECTION directives, type the LOAD command or #LOAD function with the file name of your library file. To load the file whenever you log on, include the LOAD or #LOAD call in your TACLCSTM file. The load process includes partial interpretation of statements, making subsequent invocations more efficient.

Creating a Segment File

Segment files are memory-mapped files that can be loaded into an extended memory segment. When you attach a segment file, you load it into memory so that TACL can gain immediate access to your macros, routines, and other variables. Segment files provide efficient storage for commonly used macros and routines.

The Default TACL Segment File

Whenever you log on, TACL creates a private segment file to hold the variables in the root (:) directory; this file is called the TACL default segment file. TACL then creates the directory UTILS and attaches the segment file TACLSEGF to it for shared access. TACLSEGF contains directories for all the software RVU products that have TACL programs and that are available on your system. Each TACL command is stored as :UTILS:TACL: command. For additional information about directories, see Section 6, The TACL Environment.

User-Defined Segment Files

To create a segment file, you load a library file into a segment. Unlike a library file, you do not need to load it again unless you change its contents. After the contents of the file are in the segment file, the ATTACHSEG and USE commands establish access to the variables in the segment. If the segment has not been detached since you last logged on, only the USE command is needed.

You can add your own library of variables to the default segment file established by TACL at logon time, or you can create your own personal segment file. If you load your variables into the default segment file, you must do so every time that file is recreated. If you have your own segment file, you direct TACL to attach the file to a directory variable and to use that directory for reference to the file. Because each load performs

partial code interpretation, the greater the number of variables you use regularly, the more efficiency you can gain by using personal segment files instead of loading the variables into the default segment.

Sharing a Segment File

A segment file can be restricted to your TACL process or shared among multiple TACL processes. You can read and modify variables in a private segment file; variables in a shared segment file can be read but not modified.

Commands and Built-In Functions That Manage Segment Files

There are several TACL commands that manage segment files:

- CREATESEG creates and initializes a new segment file.
- LOAD loads the contents of a library file into a segment file.
- ATTACHSEG gains access, in either private or shared mode, to an existing segment file.
- DETACHSEG detaches a segment file and removes its contents from memory.
- You can detach any segment file except the default segment file. After a segment file has been detached by all TACL processes that had attached it, the segment is available for use again as either a private or shared segment file.
- SEGINFO displays information about all segment files that your TACL currently has attached. The SEGINFO display includes a use count, which is a count of variables that are being used by your TACL for process I/O, in the use list, or that are in your home directory.

There are also several built-in functions that manage segment files: #SEGMENT, #SEGMENTINFO, and #SEGMENTVERSION return information about segment files, and #SEGMENTCONVERT converts a segment file from C20 formats to formats of segment files created on an RVU of TACL released at C20 or after. (Later RVUs of TACL are different from earlier RVUs because of the addition of the international character set).

For additional information about the creation and use of segments, see individual command and function descriptions in Section 8, UTILS:TACL Commands and Functions and Section 9, Built-In Functions and Variables respectively.

Creating and Accessing a Segment File: An Example

This example shows one method for setting up a segment file; it demonstrates most of the commands and functions that relate to segment files, the home directory, the use list, and #PMSEARCHLIST.

```
== Create and attach the segment
CREATESEG mysegfil
ATTACHSEG PRIVATE mysegfil :mydir

== Change the name of the home directory for the next few
== commands
HOME :mydir
== Load library files into the segment file
SINK [#LOAD /KEEP 1/ filename]
SINK [#LOAD /KEEP 1/ filename] == As many of these as you
SINK [#LOAD /KEEP 1/ filename] == need to load all libraries

== Change the home directory back to the root directory
HOME

== Delete :mydir and close the segment file. This causes the
== new contents to be written to the segment file.
DETACHSEG :mydir
```

To modify the segment file, close the segment file (with a DETACHSEG command), and then follow the previous steps, entering commands to create, delete, or change variables in place of the #LOAD commands.

These commands show how to provide shared access to the new segment file:

```
== Attach the segment file to a directory
ATTACHSEG SHARED mysegfil :mydir
#SET #USELIST [#USELIST] :mydir
```

To view your current use list, use the ENV command.

These lines of code, when included in your TACLCSTM file, attach a segment file whenever you log on:

```
?TACL MACRO
#PUSH taclversion
#SETMANY taclversion , [#TACLVERSION]
[#IF NOT %1% |THEN|
  [#CASE [taclversion] == To make TACLCSTM work with any TACL
  | t9205b20 t9205b30 t9205b40 |

***

Commands to load your libraries (for older TACL versions)

***

  | t9205c00 t9205c10 |
  ATTACHSEG SHARED mysegfil :mydir
  | OTHERWISE |
```

```
  #OUTPUT Unknown TACL version: [taclversion]
] == End CASE

] == End IF
[#CASE [taclversion]
| t9205c00 t9205c10 | USE :mydir
| OTHERWISE |
]
#POP taclversion
#SET #PMSEARCHLIST $SYSTEM.SYSTEM [#DEFAULTS]
```

Note. To provide shared access to a segment file, you must specify shared when you attach the file.

Releasing a Segment File

Unless you use the SEGRELEASE option with the LOGOFF command, TACL delays detaching and purging its default segment file-an advantage if you are the next user to log on. If a different user logs on, TACL detaches and purges that default segment file and creates and attaches a new one.

Note. If you log on at a terminal other than your usual one, or through a modem, use the SEGRELEASE option when you log off. Otherwise, the TACL process keeps your segment file open until someone else logs on, and you cannot detach it and reattach it in PRIVATE mode if you decide to modify it.

If the CPU on which your TACL process is running fails while you are in the act of detaching a private segment file, the segment file may be corrupted. TACL detects corruption of segment files and prevents their reuse. If this happens to one of your segment files, you must purge it and re-create it.

In general, TACL segment files created by previous RVUs of TACL do not have to be rebuilt; however, if an attempt is made to use an incompatible segment file, TACL will give a specific error message. Rebuilding is always possible from the original libraries without change, but there is no guarantee that you will be able to decompose a segment file into its original libraries.

Handling Process Completion Information

Depending on how you run a process from TACL and how the process handles termination,

- TACL can access several types of completion information: TACL supports a STATUS option that stores an indication of why the process terminated. The possible indications are STOP, ABEND, CPU (CPU failure), and NET (network failure).
- If the process specifies a completion code, you can access the completion code.

TACL stores completion code information in two STRUCT variables:

- :_COMPLETION provides compatibility with C-series software

- :_COMPLETION^PROCDEATH supports D-series process handles

TACL defines both variables when you log on, and the variables remain unless you pop (delete) them. The default values are zeros for numeric items and spaces for text items.

TACL sets specific items within these STRUCT variables whenever you try to start a process, whenever you successfully start a process, and whenever a process you started terminates (successfully or otherwise). Whenever TACL sets these variables, their previous contents are lost. For information about the contents of these variables, see:

- C-Series :_COMPLETION Variable on page 5-17
- D-Series :_COMPLETION^PROCDEATH Variable on page 5-18

If you define your own variable named :_COMPLETION or :_COMPLETION^PROCDEATH, it should be a STRUCT variable. Each time TACL stores data in :_COMPLETION or :_COMPLETION^PROCDEATH, TACL sets the STRUCT to default values. If the STRUCT is shorter than the data that needs to be stored in it, TACL discards the extra data. If the STRUCT is longer than the data to be stored in it, the extra space remains set to default values.

C-Series :_COMPLETION Variable

TACL updates completion code information in the variable :_COMPLETION, if it exists. The standard TACL software file TACLSEGF defines :_COMPLETION as follows:

```
[#DEF :_completion STRUCT
  BEGIN
  INT messagecode;
  CRTPID process;
  INT headersize VALUE 14;
  INT4 cputime;
  INT jobid;
  INT completioncode;
  STRUCT internal;
    BEGIN
    INT terminationinfo;
    SSID subsystem;
    END;
  STRUCT external REDEFINES internal;
    BEGIN
    BYTE group;
    BYTE user;
    CRTPID process;
    END;
  INT textlength;
  CHAR text(0:79);
  END;
]
```

If you incur a syntax error while trying to start a process, TACL sets MESSAGECODE to 0, COMPLETIONCODE to 4, and TERMINATIONINFO to 0.

If a process fails to start, TACL sets MESSAGECODE to 0, COMPLETIONCODE to 4, and TERMINATIONINFO to an error code that describes the failure. For more information, see the #NEWPROCESS Built-In Function on page 9-265.

If you start a process successfully, TACL sets MESSAGECODE to 0, COMPLETIONCODE to 0, and TERMINATIONINFO to 0.

When TACL receives a STOP or ABEND message from a D-series process, TACL stores the termination information in the :_COMPLETION variable:

- TACL tries to convert a PROCDEATH system message to a C-series STOP or ABEND system message. Note, however, that if the message represents an unnamed high PIN process, the message will not fit in :_COMPLETION. In this case, TACL fills :_COMPLETION with zeros for numeric items and spaces for text items.
- The PIN in the process identifier field is set to 255 for any high PIN value.

TACL stores the name of the terminating process in :_COMPLETION:PROCESS and stores the name of the process that requested the termination in :_COMPLETION:EXTERNAL:PROCESS. If, however, the requesting process has stopped or is not named, TACL cannot access its name. In this case, TACL sets :_COMPLETION:EXTERNAL:PROCESS to spaces. You can use CPU and PIN information instead of the process name.

For more information about MESSAGECODE and other definitions and for an example of TACL statements that process completion codes, see the *TACL Programming Guide*.

D-Series :_COMPLETION^PROCDEATH Variable

A D-series PIN does not fit into a CRTPID field, so D-series TACL stores D-series completion information in the :_COMPLETION^PROCDEATH variable, if it exists. D-series TACL receives PROCDEATH (-101) messages instead of STOP and ABEND messages, and saves any PROCDEATH messages in the :_COMPLETION^PROCDEATH variable, if it exists.

New D-series TACL applications should use the :_COMPLETION^PROCDEATH for completion information.

If a requesting process is named and started on another processor and if that processor fails, TACL still receives the PROCDEATH system message to fill in the completion information. If the requesting process is unnamed, however, TACL does not receive the PROCDEATH system message and the completion information is not updated. The requesting process should be named to make sure the completion information is updated. For more information on these limitations, see the *Guardian Procedure Calls Reference Manual*.

The TACL software file, TACLSEGF, defines :_COMPLETION^PROCDEATH as follows:

```
[#DEF :_completion^procdeath STRUCT
  BEGIN
  INT z^msgnumber;
  STRUCT z^base
```

```
          REDEFINES z^msgnumber;
        BEGIN
        CHAR byte(0:1);
        END;
      PHANDLE z^process^handle;
      INT4 z^cputime;
      INT z^jobid;
      INT z^completion^code;
      INT z^termination^code;
      INT z^killer^craid;
          REDEFINES z^termination^code;
      SSID z^subsystem;
      PHANDLE z^killer;
      INT z^termtext^len;
      STRUCT z^procname;
        BEGIN
        INT zoffset;
        INT zlen;
        END;
      INT z^flags;
      INT z^reserved(0:2);
      STRUCT z^data;
        BEGIN
        CHAR byte(0:111);
        END;
      STRUCT z^termtext
             REDEFINES z^data;
        BEGIN
        CHAR byte(0:111);
        END;
      STRUCT z^procname^
             REDEFINES z^data;
        BEGIN
        CHAR byte(82:193);
        END;
      END;
    ]
```

If you incur a syntax error while trying to start a process, TACL sets `z^completion^code` to 4.

If a process fails to start, TACL sets `z^completion^code` to 4.

If you successfully start a process, TACL sets `z^completion^code` to 0.

The field `:_completion^procdeath:z^procname:zoffset` contains the byte offset of the process name. The process name will always be within the substructure `z^data`, so the offset will always be between 82 and 193.

When a process is named, `:_completion^procdeath.z^procname.zlen` is greater than zero.

Otherwise, `:_completion^procdeath.z^procname.zlen` equals zero and `:_completion^procdeath:z^data` contains spaces.

To access the process name:

```
#PUSH proc^offset proc^len proc^lwa procname
#SET proc^len &
  [:_completion^procdeath:z^procname:zlen]
[#IF proc^len > 0 |THEN|
  #SET proc^offset &
    [:_completion^procdeath:z^procname:zoffset]
  #SET proc^lwa [#compute proc^offset+proc^len-1]
  #SET procname &
    [:_completion^procdeath:z^procname^:byte([proc^offset]:&
    [proc^lwa])]
]
```

To access the termination text:

```
#PUSH termtext^len termtext^lwa termtext
#SET termtext^len &
  [:_completion^procdeath:z^termtext^len]
[#IF termtext^len > 0 |THEN|
  #SET termtext^lwa [#compute termtext^len-1]
  #SET termtext &
   [:_completion^procdeath:z^termtext:byte(0:[termtext^lwa])]
]
```

Interactive Display of Completion Code Information

An interactive TACL displays completion code information whenever one or more of these statements are true:

- PMSG is ON.
- :_COMPLETION:MESSAGECODE is -5 (STOP) and :_COMPLETION:COMPLETIONCODE is not 0 and not 6 (stopped externally).
- :_COMPLETION:MESSAGECODE is -6 and :_COMPLETION:COMPLETIONCODE is not 5 and not 6 (stopped externally).
- :_COMPLETION:TERMINATIONINFO is not 0 and :_COMPLETION:COMPLETIONCODE is not 6 (stopped externally).
- :_COMPLETION:TEXTLENGTH is not 0.

Handling TACL Errors

TACL can generate several types of errors. Table 5-1 lists the types of errors, a sample display, and a description and action for each type of error.

Table 5-1. Error Types

Error	Example	Description
Syntax error	Expecting an existing variable, unqualified	In this example, TACL required a variable. In general, TACL required one or more types of elements and did not find them. The program stops executing at the statement indicated in the message. The message includes the command line in question, a pointer that indicates where the potential problem is, and the type of argument or value that TACL expects to find in the indicated position.
TACL error	*ERROR* Security Violation	The program used a construct incorrectly or supplied an argument that would not work in the specified manner. For example, if you set #OUT to a file that you do not have write access to, you receive *ERROR* Security Violation. The program stops unless it includes an exception handler and an _ERROR clause in an associated #FILTER statement. (For information about exception handlers, see Section 3, “Developing TACL Routines,” in the *TACL Programming Guide*.)
Error During a function call	None	A built-in function such as #PURGE or #RENAME can return file system or operating system errors. The program continues. To check for these errors, store the contents of the result into a variable and check the result before continuing.

Table 5-1. Error Types

Error	Example	Description
Error defined by program	None	The program detected a condition defined as an error in the local TACL environment; this error could be a volume name that should not be accessed by a particular user. Action depends on the code. The program can exit or raise a user-defined exception, which causes TACL to invoke an exception handler if one is defined.
Fatal error	ABENDED: $TCL2 CPU TIME 0:00:00.013 Termination Info 14 TACL fatal error: Couldn't open TACL IN	ABENDED: $TCL2 CPU TIME 0:00:00.013 Termination Info 14 TACL fatal error: Couldn't open TACL IN TACL encountered a problem with an external file, memory, or other resource, and could not continue. If, for example, TACL cannot find its IN file at startup time, it produces this error. Termination info reports the error that TACL received. TACL abends or stops itself.
Internal error	ABENDED: $TCL4 CPU TIME 0:00:00.052 Termination Info 18 TACL internal error: Initialize.100	TACL encountered an internal error condition and could not continue. TACL abends or stops itself.

TACL programs can check for function call errors immediately after a function call. To handle other types of errors, see the discussion about exception handlers in the *TACL Programming Guide*.

TACL can generate one type of EMS error: Error 66, an I/O error.

6 The TACL Environment

This section describes these operational topics:

- Files that are required and recommended for TACL operation
- Starting a TACL process and subordinate processes
- Using TACL directories
- Running TACL as a background process

For information about the use of TACL with other subsystems, see the *TACL Programming Guide*. Section 7, Summary of Commands and Built-In Functions, provides an overview of TACL commands and functions.

Installation Instructions

TACL is now also delivered with HIGHPIN set to ON. To get TACL with HIGHPIN set to ON, follow the below steps:

1. Run this command:

   ```
   FUP RENAME <install-vol>.<SYSnn>.TACL, TACLLOW
   ```

2. Run this command:

   ```
   FUP DUP <install-vol>.ZTACL.TACLH,<install-vol>.<SYSnn>.TACL
   ```

3. Restart all TACL processes..

Note. From this SPR the ZTACL subvol will be available, under which the TACLH file for HIGHPIN is present.

TACL Software RVU Files

TACL uses these files:

- TACL, a program file that is usually stored at $SYSTEM.SYS*nn*.TACL.
- TACLINIT, an edit file that resides on the same subvolume as the TACL file.
- TACLCSTM, an edit file that provides customization for your personal TACL environment. The file starts with a ?TACL MACRO directive and resides in your default subvolume.
- TACLLOCL, an edit file that is invoked by TACL to perform customization of the TACL environment for all TACL users on a system. The file starts with a ?TACL MACRO directive and resides on the same subvolume as the TACL file.
- TACLSEGF, a segment file that contains TACL commands and the code for all other products (that are part of a software RVU) on the system that include TACL programs.

- TACLBASE, an edit file that contains the same functionality as TACLSEGF. This file resides on the same subvolume as the TACL file. Along with providing functionality, TACLBASE provides a readable source of examples of TACL programs.
- TACLCOLD, a segment file that TACL uses when running as the coldload command interpreter. TACL creates this file or reuses it as a way of reducing the chance that the coldload TACL will fail due to lack of disk space at startup.
- CPRULES0 and CPRULES1, which define the character set in use by TACL. CPRULES0 is the default set.

In addition to the preceding list of files, there are utility programs that assist TACL in performing certain operations. Each program is in a separate program file in $SYSTEM.SYS*nn* or $SYSTEM.SYSTEM. These programs:

- Perform privileged operations, such as establishing user IDs and passwords or reloading processors
- Must be licensed for use by nonprivileged users
- Can run only on the local system

Utility programs are listed in Section 8, UTILS:TACL Commands and Functions.

TACL requires the TACL program file and the TACLINIT file. If TACLBASE and TACLSEGF are not present, TACL can operate, but will provide only built-in functions and variables.

If TACLLOCL is missing, an unfriendly user could create a file with that name and place harmful commands (such as PURGE commands) in it. To minimize this danger, TACL creates a dummy TACLLOCL file whenever any user logs on and TACLLOCL is not present. TACL secures the file to “NUUU” and gives it to the owner of the TACL process itself.

A similar danger exists if your TACLCSTM file is missing, so TACL creates a dummy TACLCSTM file when you log on and TACLCSTM is not present. TACL secures the file to your default security and gives ownership of the file to you.

Starting a TACL Process

To start an interactive TACL process, run TACL with the IN and OUT files specified as the same name, that of a terminal device. System management usually starts a TACL process pair for each terminal, using the RUN or TACL command or the #NEWPROCESS built-in function.

Note. All TACL built-ins are executed by the TACL process, which runs only on the node where it was started, regardless of any SYSTEM commands that are issued. To execute a built-in command on another system, you must start a new TACL process on that system.

A TACL process:

- Can run at a high PIN
- Can create a high-PIN process
- Can be created by a high-PIN process
- Can communicate with a high-PIN requester
- Can communicate with a high-PIN server
- Recognizes high-PIN process identifiers
- Does not default to run at a high PIN

Processes that run at high PINs cannot open and write to processes that do not allow high-PIN openers. For example, a TACL process that runs at a high PIN cannot open a process on a C-series node. Similarly, a TACL process on a C-series node cannot open a high-PIN process. This limitation applies to any operation that accesses process identifier information, such as alteration of priority. If you start a process and try such communication with the new process, the new process terminates.

Logging On

To access a TACL process, you must log on. Before you log on, the only commands TACL accepts are LOGON, FC, !, PAUSE, and TIME. If the system runs Safeguard software, Safeguard can be set up so that it authenticates your user ID and requests that TACL start logged-on. In this case, you do not need to log on to TACL.

TACL Initialization

When you log on, TACL creates a private segment file for you and invokes TACLINIT (usually $SYSTEM.SYS*nn*.TACLINIT). TACLINIT begins initialization of TACL and then searches in the same subvolume for the TACLSEGF segment file, which it attaches and makes available under the directory :UTILS:TACL. Directories are described in Using Directories on page 6-9.

TACL searches for TACLLOCL first in the SYS*nn* of the TACL PROGRAMFILE, and then in $SYSTEM.SYSTEM.

You can ensure that there is always a system load recovery path by placing a TACLLOCL in each SYSnn. If a TACLLOCL change causes any problem, reload the system from the original SYS*nn*.

This method also offers a superior organizational scheme, because a TACLLOCL for different system images often needs to have different TACL statements. Rather than having complex version condition checking, each system image version can now have its own TACLLOCL file.

If it finds the file, TACL invokes it as a macro. At this point, if your default subvolume contains a TACLCSTM file, TACL invokes it as a macro. Your TACLCSTM file, which

contains any commands you supply, can include a request to load all your personal command definitions.

TACL passes one argument to TACLCSTM, indicating whether the loading of TACLBASE was suppressed because you are the most recent user to log off from this TACL and are now logging back on. (The same segment is still in effect.) The argument is nonzero if loading was suppressed, and zero if it occurred.

The TACLLOCL of the installation or your individual TACLCSTM can also create directories for additional segment files.

CPRULES0 and CPRULES1 define the character set in use by TACL. CPRULES0 is the default set. When starting a TACL process, if #CHARACTERRULES is empty after TACLLOCL has been invoked, the TACL process looks for CPRULES0. TACL searches for CPRULES0 in $SYSTEM.SYSTEM. If CPRULES is not found there, then TACL searches the same volume and subvolume in which the TACL program file resides (usually, $SYSTEM.SYS*nn*). If a CPRULES file does not exist when a user logs on and the TACL process tries to access a CPRULES file, the TACL process issues a warning that it is using its own set of rules (that are encoded in the TACL program file).

TACL sets the initial terminal state to:

- ECHO on
- Single spacing
- Post spacing
- Conversational mode

Note. If a backup TACL process takes over at a later time, TACL initializes the terminal state to the preceding values unless the TACL process has a descendent process that continued to run during the processor switch. In this case, TACL does not initialize the terminal state values.

After your TACL process completes initialization and you log on, you can:

- Issue TACL commands or other statements (supplied in :UTILS:TACL and stored in the TACLBASE file) by typing the command
- Store TACL programs in files and invoke the programs when needed, in one of three ways:
  - Store a text, macro, or routine variable in its own file and invoke it by file name
  - Store one or more programmatic variables in a library file and issue a LOAD command or #LOAD call to make the variables accessible to TACL
  - Save one or more programmatic variables in a segment file and issue an ATTACHSEG command to bring the functions into memory

For more information about these procedures, see Section 5, Statements and Programs.

Starting New Processes

To start a new process from a TACL process, use the RUN command or the #NEWPROCESS built-in function, or specify the program file name (an implicit RUN command). If you use an implicit RUN command, ensure that the #PMSEARCHLIST setting includes the location of the program file. For more information, see Section 9, Built-In Functions and Variables.

You can request to run a process at a high or low PIN. Several options affect whether a new process runs at a high or low PIN, including the TACL HIGHPIN variable, the BINDER HIGHPIN option, and the TACL RUN and #NEWPROCESS /HIGHPIN ON/ option.

These conditions must be met before TACL can start a process at a high PIN:

- The HIGHPIN option for the code file must be enabled at compile or bind time
- Either the RUN /HIGHPIN ON/ option or the #HIGHPIN built-in variable (or both) must be ON
- There must be a high PIN available

Table 6-1 describes how TACL resolves each combination of HIGHPIN settings. A dash (-) indicates that the option is not specified.

Table 6-1. Results of HIGHPIN Settings

Parent TACL Runs at a	BINDER HIGHPIN Option	HIGHPIN Variable	RUN or #NEWPROCESS HIGHPIN Option	New Process Runs at a
Low PIN	OFF	OFF	-	Low PIN
Low PIN	OFF	OFF	OFF	Low PIN
Low PIN	OFF	OFF	ON	Low PIN
Low PIN	OFF	ON	-	Low PIN
Low PIN	OFF	ON	OFF	Low PIN
Low PIN	OFF	ON	ON	Low PIN
Low PIN	ON	OFF	-	Low PIN
Low PIN	ON	OFF	OFF	Low PIN
Low PIN	ON	OFF	ON	High PIN*
Low PIN	ON	ON	-	High PIN*
Low PIN	ON	ON	OFF	Low PIN
Low PIN	ON	ON	ON	High PIN*
High PIN	OFF	OFF	-	Low PIN
High PIN	OFF	OFF	OFF	Low PIN
High PIN	OFF	OFF	ON	Low PIN
High PIN	OFF	ON	-	Low PIN
High PIN	OFF	ON	OFF	Low PIN
High PIN	OFF	ON	ON	Low PIN
High PIN	ON	OFF	OFF	Low PIN
High PIN	ON	OFF	OFF	Low PIN
High PIN	ON	OFF	ON	High PIN*
High PIN	ON	ON	-	High PIN*
High PIN	ON	ON	OFF	Low PIN
High PIN	ON	ON	ON	High PIN*

* The process runs at a high PIN if a high PIN is available.

To check the setting of the #HIGHPIN variable from an interactive terminal, type SHOW HIGHPIN. To check the setting of the BINDER HIGHPIN option for an object file (type 100 or 700), enter:

```
BIND; SHOW SET * FROM file-name
```

For additional information about the BINDER HIGHPIN option, see the *Binder Manual*.

Customizing the TACL Environment

TACL supports two types of customization:

- Personal customization for one user, through the use of the TACLCSTM file
- Local customization, through use of the TACLLOCL file

Customization affects all uses of TACL, not just interactive TACL sessions started from TELNET or SSH. Customization of TACLCSTM and TACLLOCL must allow server programs to start TACL processes without interference.

Standard server programs using background TACL processes do not tolerate interactive queries when TACL starts. Arbitrary redefinition of standard TACL commands or environment might also cause difficulties.

Examples of products that start background TACL processes include some HP OSM program files, NonStop Essentials, and the SeeView Server Gateway. These products start TACL processes either on behalf of a command from a human-computer interface or for unattended background activities.

Customization of a particular user ID's TACLCSTM for security auditing might require special care. A TACLCSTM file for a particular user can check the ancestor process of the TACL process for special exceptions. For an example of this customization for a particular user in the SUPER group, see the *OSM Configuration Guide*.

Personal Customization

To customize your TACL environment, you can add the following to your TACLCSTM file:

- Function-key definitions, command aliases, macros, or routines in edit-format files.
- LOAD commands or #LOAD functions to load files into variables in your home directory (the root directory, by default). When TACL invokes the TACLCSTM file as you log on, it loads the files.
- ATTACHSEG and USE commands to gain access to a segment file. Segment files are described in Section 5, Statements and Programs.
- Commands such as TIME or SETPROMPT, or your own programs, to be invoked when you log on.

TACLCSTM can also change your saved defaults by assigning volume and subvolume names to the variable :UTILS_GLOBALS:TACL:_DEFAULTS_INITIAL.

Local Customization

The TACLLOCL file provides a mechanism for environment customization for all users on a given system (you might think of TACLLOCL as the system manager's version of TACLCSTM). This example shows a sample TACLLOCL file:

```
?TACL MACRO
== TACLLOCL sets up variables, macros needed for operations
RUN $system.system.opertool == defines _oper^tools variables
[#IF %1% |THEN|
  == A fast logon
|ELSE|
  == A slow logon
  SINK [#LOAD /KEEP 1/ [_oper^tools].macrolib]
]
```

In the preceding example, the dummy argument %1% refers to an argument provided by TACL when you log on; you can use this same construct in your TACLCSTM file. The argument is true (a nonzero value) if the default segment file containing TACL variables exists, false (zero) if TACL created a new segment file when you logged on. If the segment file exists, loading library files is unnecessary.

Managing the BREAK Key

When you first log on, TACL initializes #BREAKMODE to POSTPONE (see #BREAKMODE Built-In Variable on page 9-36). The standard TACL logon support sets #BREAKMODE to ENABLE just before invoking your TACLCSTM. This prevents you from breaking out of operations needed to configure your TACL when you log on, yet allows you to break out of TACLCSTM if it becomes necessary.

Thereafter, you can set #BREAKMODE to enable, disable, or postpone the action of the BREAK key as needed to ensure that any given sequence of TACL statements can be free of interruption. This feature should be used with caution, however, because it removes user control of the BREAK key. If you use a TACL with a customized version of TACLINIT or TACLSEGF, you must ensure that #BREAKMODE is set to ENABLE at some point during initialization if you want users to be able to use the BREAK key to interrupt TACL processing.

Security

There exists a potential breach of security if other TACL users open your TACL process. To limit access to your TACL process, use the #TACLSECURITY built-in variable, described in #TACLSECURITY Built-In Variable on page 9-400.

Command Interpreter Monitor Interface (CMON)

The purpose of a command interpreter monitor ($CMON) process is to control and monitor:

- Logons and logoffs, including illegal logon attempts (LOGON and LOGOFF commands)
- Addition or deletion of users (ADDUSER, DELUSER programs)
- Alteration of priorities at execution time (ALTPRI command)
- Process startups (RUN command and #NEWPROCESS built-in function)

If a $CMON process exists on a system, TACL communicates information to $CMON when the above activities are requested.

TACL sends these types of messages to $CMON:

- A CMON configuration message, which establishes the TACL configuration for the new process or user.
- A pre-LOGON message, which allows $CMON to invoke additional security provisions (for example, requiring a user to log on under one ID before being able to log on under some other ID).

If the $CMON process is not running, TACL uses stored defaults.

For a description of the communication between TACL and a command interpreter monitor process, see the *Guardian Programmer's Guide*.

Using Directories

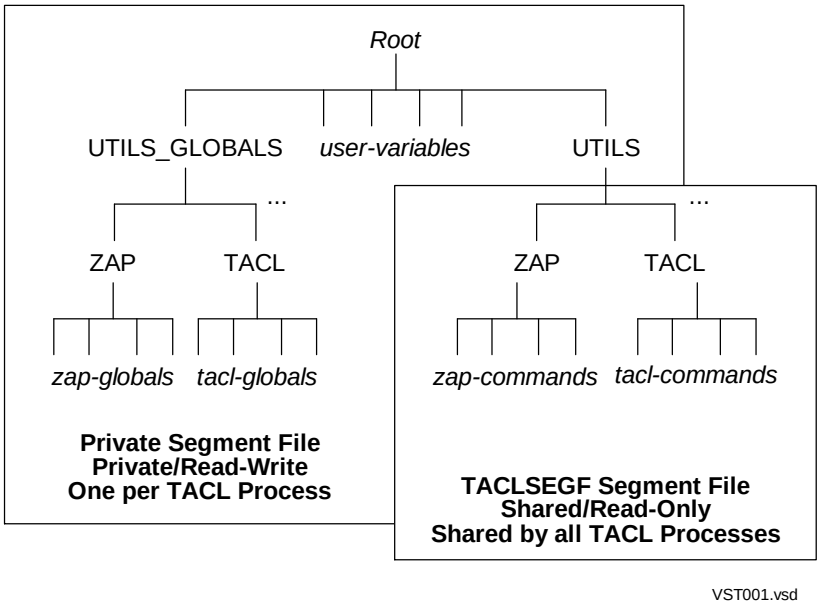
TACL organizes its variables according to purpose or product; it keeps your variables separate from TACLBASE variables (commands) and helper variables that are used by the TACLBASE variables and are not intended for direct use. This organization called a directory, and is hierarchical, similar to a tree. The root of the tree is the :directory.

Directories organize variables in a hierarchy and take advantage of operating system features that support segment files.

A Sample Directory Structure

Figure 6-1 on page 6-10 demonstrates the structure of variables and segment files in a running TACL process. ZAP is an example of an application product (that are part of a software RVU); each application product that releases TACL programs has a directory equivalent to those shown for ZAP. Any segment file that you create would be attached among the user-vars.

Figure 6-1. TACL Segment File and Directory Relationships



Creating Your Own Directories

You can create your own directories of variables. For more information, see Section 4, Variables. For information about creating and attaching segment files, see Section 5, Statements and Programs.

Directories Supplied by TACL

The UTILS directory exists in the root and contains directories for all application products (that are part of a software RVU) on your system that use TACL, referenced as :UTILS: product. If a product includes or consists of subproducts, each subproduct can be considered a product in its own right and be given its own directory in :UTILS.

The :UTILS:TACL directory contains all the TACL commands, including command-interpreter commands and additional commands, programs, and functions used by TACL. For example, the :UTILS:TACL variable VOLUME refers to the command named VOLUME that is executable by the TACL program.

There are also some primitive functions in :UTILS:TACL. The name of each of these functions begins with an underscore (_), and each returns a value. Primitive functions differ from commands, which primarily perform actions, in that the returned value, not the action, is the primary result.

Other products that are part of the same software RVU use another directory in the root, UTILS_GLOBALS, if they include TACL programs and need to maintain writable global TACL variables. This use of directories achieves the following:

- Minimizes the potential for naming conflicts among variables from various sources

- Provides for read-only sharing of variables among TACL users
- Removes the need for loading each library of variables before each use
- Removes the need for predicting the maximum segment file size a given TACL user will require

Other application products (that are part of a software RVU) can also create directories. TACL requires you to specify where to put the associated variables or, by default, stores them in your current home directory. (To store variables in your home directory, your home directory must be writable.) The documentation for each product explains how to access and use these files.

Avoiding Naming Conflicts With TACL

All temporary variables created by TACL begin with colon-circumflex (:^). TACL stores these variables in the root directory, which is always writable, and guarantees that these temporary variables will never conflict with any of your own variables.

The TACLINIT file creates a directory called :UTILS_GLOBALS in the default segment file; each product that needs global, writable variables creates its own directory and variables within the :UTILS_GLOBALS directory. Such variables are named :UTILS_GLOBALS: product: var-name.

All TACL variables included with an application product (that is part of a software RVU), and intended solely for use by that product, are put into subordinate directories in the directory of the product; the name of each variable of this type begins with a circumflex (^). This convention allows you to list the directory of a product without memorizing the names of private utilities.

To avoid conflicts with TACL and application products using TACL, adhere to these rules:

- Do not create variables whose names begin with a circumflex (^) and never use, in any way, such variables.
- Do not create or use variables whose names begin with an underscore (_), except where specifically permitted as a feature of an application program.
- Do not create any variables under :UTILS.
- Do not create any variables under :UTILS_GLOBALS, except where specifically permitted as a feature of an application program.
- Do not push or pop :UTILS or :UTILS_GLOBALS.
- If you modify the use list, ensure that your use list always includes certain directories necessary for the correct operation of the application program. The USE command automatically does this for you. The list of necessary directories depends on the application software version and must not be hard coded in your TACL programs.

_EXECUTE Variables

If you try to invoke a variable that is itself a directory, TACL searches that directory for a variable named "_EXECUTE" and invokes that variable instead. The _EXECUTE variable provides an automatic mechanism for special initialization activities such as:

- Displaying a banner
- Establishing a frame
- Pushing the use list and altering it
- Creating variables
- Starting servers
- Opening files
- Establishing an exception handler

For example:

```
?SECTION _execute ROUTINE
PATHCOM $XXPM;RUN XX
```

Some application products allow you to enter TACL commands while you are using them. This flexibility allows you to invoke another product without leaving the first product.

Running a TACL Process in the Background

Another way to start a TACL process is to start one as a background process. A background process is a process that runs with the NOWAIT, INV, or INLINE option. A background process allows you to perform noninteractive tasks without interfering with your interactive work. This type of TACL process starts running and can wait for commands indefinitely; the background process does not prompt for input.

Initializing TACL and Specifying Input

To create a background TACL, specify a file name as the IN file. You can, for example, specify an EDIT file with a set of TACL commands (and without a ?SECTION or ?TACL directive) as the IN file:

```
TACL /IN file-name, NOWAIT, NAME $TCL2/
```

To send commands interactively to the background process, use the INV or INLINE run options. To execute a routine from the background process, you can supply an IN file that loads and runs the routine.

When a background TACL starts, you do not need to log on to it; it automatically starts up logged on as the user who started it.

Default Files

Instead of using your saved defaults, a background TACL takes on the same file-name defaults as those of the process that started it. This procedure occurs as follows:

1. TACL saves the inherited defaults in the variable :UTILS_GLOBALS:TACL:_DEFAULTS_INITIAL and switches to your saved defaults (saved under your user ID on the system where the background TACL is running). TACL saves a copy of those defaults in the variable :UTILS_GLOBALS:TACL:_DEFAULTS_SAVED.
2. TACL invokes the specified TACLCSTM file (if any) in the same way and with the same defaults as if you logged on explicitly. To specify a TACLCSTM file, use the ASSIGN command in the process that starts the background TACL:

   ```
   ASSIGN TACLCSTM, file-name
   ```

   If *file-name* is not fully qualified, ASSIGN assumes the defaults in existence for the process that issued the ASSIGN command (not the process receiving it).

   To omit the TACLCSTM file, enter this; the comma indicates the absence of the file-name field:

   ```
   ASSIGN TACLCSTM,
   ```

   If you do not issue an ASSIGN TACLCSTM command, the background TACL uses your current TACLCSTM file by default.
3. After it has invoked TACLCSTM, TACL sets the defaults to the value in the variable :UTILS_GLOBALS:TACL:_DEFAULTS_INITIAL. If you need to establish a particular set of initial defaults, your TACLCSTM should contain coding to set :UTILS_GLOBALS:TACL:_DEFAULTS_INITIAL accordingly.

You can read the variables in the directory :UTILS_GLOBALS (and its subdirectories), but do not change them, except as directed in application-product documentation.

7

Summary of Commands and Built-In Functions

TACL provides three types of standard capabilities:

- Commands, which are implemented as TACL variables
- Built-in functions, which are not implemented as TACL variables but are accessed in the same manner as TACL variables
- Built-in data variables

This section provides an overview of these standard capabilities, followed by tables that summarize commands, functions, and variables by functional group. All these constructs can be used interactively or in a TACL program. When there are two constructs that support the same function, however, use the command for interactive work and the built-in function for programmatic work:

- Commands typically display results. Commands provide information about files, allow you to check the status of processes, gather information about your own processes and those of other users, and perform other types of work. The RUN command runs other programs such as the File Utility Program (FUP).

  The :UTILS:TACL directory contains the set of standard TACL commands.
- Built-in functions and variables provide information that can be used by a program. They return information and status in the form of a result. The set of built-in functions and built-in variables provide the fundamental, fixed set of TACL functionality.

TACL Commands

When you log on to your system, you use TACL commands. These commands include:

RUN	Runs a process
STATUS	Displays information about one or more running processes
WHO	Describes the current environment

Commands are intended for interactive use; although you can use them in TACL programs, commands do not typically return as much status or error information as do functions. The commands generally display a result.

TACL commands are interpreted. Each command is a TACL variable. All commands call TACL built-in functions.

The :UTILS:TACL directory contains all the TACL commands, including command-interpreter commands and additional commands, programs, and functions used by

TACL. For example, the :UTILS:TACL variable VOLUME refers to the command named VOLUME that is executable by the TACL program. For more information about directories, see Section 6, The TACL Environment.

Most :UTILS:TACL commands are available to every user. Some commands and some utility programs, however, are restricted so that only certain users can execute them. Restricted commands and programs can be used by these two groups:

- Group managers (users who have a user ID of *n*,255)
- Super-group users (users who have a user ID of 255,*n*)

(The terms “user ID” and “group ID” are described in the *Guardian User’s Guide*.) The command descriptions in Section 8, UTILS:TACL Commands and Functions indicate whether a command or program is restricted or not.

The super ID (a user who has the user ID 255,255) can execute any command or program reserved for group managers or super-group users.

Built-In Functions

In addition to the :UTILS:TACL commands, TACL provides built-in functions and built-in variables that can be used for the construction of macros and routines.

TACL built-in functions cannot be changed. Built-in functions provide the basic elements of TACL on which all other features, including TACL commands, are based. Where built-in functions and TACL commands provide the same functionality, built-in functions provide more error information and are slightly faster than commands. Built-in function names start with a number sign (#), ensuring that the names you choose for your macros or routines do not conflict with the names of built-in functions and variables. Examples include:

#COMPAREV	Compares one variable with another
#OUTPUT	Writes data to an output file
#PROCESSINFO	Returns information about a process

Built-in functions also provide flow control, such as loop control and exit mechanisms. To view a list of built-in functions, use the #BUILTINS built-in function.

Some built-in functions must be used within routines. These built-in functions (for example, #ARGUMENT, #MORE, and #REST) provide the mechanism by which routines evaluate their arguments and return their results. Unlike a macro, the result of a routine is not the text of the routine itself. A routine computes a result string to replace its invocation. A routine invokes #RESULT one or more times to produce a nonempty result.

If you type a built-in function that produces a result and you do not enclose it in square brackets, TACL automatically displays the result; for example:

```
56> #WIDTH
#WIDTH expanded to:
80
```

Built-In Variables

TACL built-in variables provide basic elements of TACL on which all other features, including TACL commands, are based. Unlike TACL commands, which can be customized or redefined, built-in variables are always available as described in this manual.

Built-in variable names start with a number sign (#). You can push them (the PUSH command and #PUSH built-in function create new top levels of variables), pop them (the POP command and #POP built-in function delete the top level of a variable), invoke them, and assign values to them, but you cannot delete them. There are certain other restrictions on their use, explained in individual function and variable descriptions in Section 9, Built-In Functions and Variables.

Examples of built-in variables include:

#OUT	The name of the OUT file used by TACL
#PMSG	The state of the PMSG flag
#MYTERM	The name of the home terminal

This example changes the TACL output file (stored in #OUT) to a spooler location to receive the list of built-in functions, then restores the output file to its previous identity:

```
96> #PUSH #OUT
97> #SET #OUT $S.#LP
98> #BUILTINS /FUNCTIONS/
99> #POP #OUT
```

You can use #PUSH (or PUSH) to save a copy of the existing contents of the top variable level and #POP (or POP) to restore the previous contents to the top of the stack. #PUSH, however, does not create such a variable, because it already exists, nor can #POP delete the variable entirely (you get a “was not pushed” error if you try to pop the first level).

In addition, pushing a built-in variable copies the pushed contents to the new top level (the built-in variable and the pushed level then have the same value). Also, many built-in variables have default values or automatically stored values. Unlike your other variables, which can be preserved from one TACL session to another, built-in variables are popped completely and reset to their default values when you log off.

There are restrictions on the use of built-in variables; for example, a built-in variable cannot be used as a variable in an #ARGUMENT function, nor can you specify an explicit variable level (for example, #DEFAULTS.-2) for a built-in variable. The

descriptions in Section 9, Built-In Functions and Variables, explain the ways in which those variables can be used.

Summary of Functionality

The remainder of this section lists commands, built-in functions, and built-in variables by functional group; the groups are:

- Obtaining help and information
- Interfacing with the operating system
- Managing the TACL environment
- Processing text in variables
- Controlling program flow
- Debugging TACL statements

Note. All TACL built-ins are executed by the TACL process, which runs only on the node where it was started, regardless of any SYSTEM commands that are issued. To execute a built-in command on another system, you must start a new TACL process on that system.

Obtaining Help and Information

Table 7-1 lists commands that provide general help and information.

Table 7-1. Informational Commands (page 1 of 2)

Command	Description
BUILTINS	Shows TACL built-in functions and variables
COLUMNIZE	Displays a list in columnar form
COMMENT	Begins comment line in TACL command file
ENV	Displays settings of TACL environmental parameters
FC	Retrieves, edits, and reexecutes lines in history buffer
FILEINFO	Displays information about files
FILENAMES	Displays names of files in a subvolume, using a file-name template
FILES	Displays names of files in a subvolume
HELP	Displays useful information about TACL
HISTORY	Displays previously issued command lines
INFO DEFINE	Displays attributes and associated values in one or more DEFINEs residing in process file segment of current TACL process
KEYS	Displays current function-key variables
LOADEDFILES	Displays the loadfiles used by a selected process
PMSG	Controls process-identifier displays of processes you create or delete

Table 7-1. Informational Commands (page 2 of 2)

Command	Description
PPD	Displays names, `cpu,pin` designations, and ancestors of processes currently running
SEGINFO	Shows information about TACL segment files
SHOW	Displays values of attributes set with SET command
SHOW DEFINE	Displays DEFINE attributes in working attribute set and their current values
STATUS	Displays status of running processes
SYSTIMES	Displays current date and time, plus date and time of last cold load
TIME	Displays current system date and time of day
USERS (program)	Displays attributes of users and groups
VARIABLES	Displays names of all your variables
VARINFO	Displays information about specified variables
WHO	Displays information about your current default settings
?	Displays a previous command line
!	Reexecutes a previous command line

Table 7-2 lists built-in functions and variables that provide general help and information.

Table 7-2. Informational Built-In Functions and Variables (page 1 of 2)

Function	Description
#BUILTINS	Examines names of TACL built-in functions
#COLDLOADTACL	Determines if TACL process is the “cold-load TACL”
#DEVICEINFO	Gets detailed information about a device
#FILENAMES	Lists file names
#GETCONFIGURATION	Obtains settings of flags that affect TACL behavior
#HELPKEY	Holds name of current help key
#HISTORY	Operates on commands in history buffer
#KEYS	Displays defined function keys
#PROCESSORSTATUS	Determines status of 16 possible processors on a given system
#PROCESSORTYPE	Determines processor type of given system or process
#SEGMENTVERSION	Determines whether segment file is C00/C10 format or newer format
#TACLVERSION	Obtains TACL product RVU
#TOSVERSION	Obtains current RVU number of the operating system

Table 7-2. Informational Built-In Functions and Variables (page 2 of 2)

Function	Description
#VARIABLES	Obtains names of all variables in your home directory
#XFILES	Implements FILES command
#XPPD	Implements PPD command
#XLOADEDFILES	Implements LOADEDFILES command
#XSTATUS	Implements STATUS command

Interfacing With the Operating System

TACL supports three types of operating system interface commands and functions:

- Handling files and devices
- Controlling processes
- Managing the system environment

Handling Files and Devices

Table 7-3 lists the commands that support file and device handling.

Table 7-3. File and Device Commands

Command	Description
ALARMOFF (program)	Turns off HP NonStop VLX system audio alarm
COPYDUMP (program)	Copies and compresses tape dump file or existing disk dump file into a disk dump file
CREATE	Creates an unstructured disk file
INITTERM	Initializes home terminal by reinstating its default SETMODE settings
LIGHTS (program)	Controls processor panel lights
PURGE	Purges (deletes) a disk file
ALARMOFF (program)	Turns off NonStop VLX system audio alarm
COPYDUMP (program)	Copies and compresses tape dump file or existing disk dump file into a disk dump file
CREATE	Creates an unstructured disk file

Table 7-4 lists the built-in functions and variables that support file and device handling.

Table 7-4. File and Device Built-In Functions and Variables

Function	Description
#CREATEFILE	Creates a file
#EOF	Sets flag so that a process receives an end-of-file after reading all data in a variable
#FILEINFO	Gets information about a file
#FILEGETLOCKINFO	Gets information about record and file locks
#IN (variable)	Holds name of IN file used by TACL
#INITTERM	Resets your home terminal to its configured settings
#INPUT	Reads information from TACL primary input file
#INPUTEOF (variable)	Holds state of INPUTEOF flag
#INPUTV	Reads information from TACL primary input file into a variable level
#LOCKINFO	Gets information about record locks
#NEXTFILENAME	Determines file following specified file
#OPENINFO	Gets information about file openers
#OUT (variable)	Holds name of OUT file used by TACL
#OUTPUT	Writes data to an output file
#OUTPUTV	Writes contents of a variable level to an output file
#PURGE	Deletes a file
#RENAME	Changes the name of an existing disk file
#REPLY	Adds text to reply if TACL IN file is $RECEIVE
#REPLYV	Adds copy of text from variable to reply if TACL IN file is $RECEIVE
#REQUESTER	Reads from and writes to files
#WIDTH (variable)	Holds value of width register
#XFILEINFO	Implements FILEINFO command
#XFILENAMES	Implements FILENAMES command

Controlling Processes

Table 7-5 lists the commands that support process control.

Table 7-5. Process Control Commands (page 1 of 2)

Command	Description
ACTIVATE	Reactivates a previously suspended process
ADD DEFINE	Creates one or more DEFINEs
ALTER DEFINE	Changes attributes of one or more DEFINEs
ALTPRI	Alters execution priority of a process
ASSIGN	Gives file attributes to logical-file descriptions used by application programs
CLEAR	Deletes attributes previously set by ASSIGN or PARAM commands
DELETE DEFINE	Removes one or more DEFINEs from process file segment (PFS) of current TACL process
EXIT	Used interactively, stops current process; used in a command file, stops execution of commands
INLECHO	Controls copying to TACL OUT file of lines sent to inline process
INLEOF	Sends end-of-file indication to an inline process
INLOUT	Controls copying to TACL OUT file of lines sent to OUT file of inline process
INLPREFIX	Establishes prefix that identifies lines to be passed to inline process
INLTO	Establishes variable to receive copies of lines sent to OUT file of inline process
PARAM	Assigns parameter value to a parameter name, or displays all current parameters and their values
PAUSE	Makes TACL stop prompting for commands and allows another process to control terminal
RESET DEFINE	Restores attributes in DEFINE working set to their initial values
RUN or RUND	Runs a program; optionally puts resulting process into debug state
SET DEFINE	SET DEFINE establishes a value for one or more DEFINE attributes in working attribute set
SET DEFMODE	Controls whether DEFINEs are enabled for current TACL process and are propagated to new processes
SET HIGHPIN	Sets the default PIN range for processes started by the current TACL process
SET SWAP	Sets swap volume for all subsequent RUN commands (unless swap volume is explicitly specified in a command)
STOP	Stops and deletes a process

Table 7-5. Process Control Commands (page 2 of 2)

Command	Description
SUSPEND	Prevents a process from running until it is reactivated by an ACTIVATE command
TACL (program)	Starts TACL process on your local system or a remote system
WAKEUP	Sets wakeup mode

Table 7-6 lists the built-in functions and variables that support process control.

Table 7-6. Process Control Built-In Functions and Variables (page 1 of 3)

Function	Description
#ABEND	Immediately terminates a process
#ABORTTRANSACTION	Aborts and backs out a transaction
#ACTIVATEPROCESS	Returns process or process pair from suspended state to ready state
#ALTERPRIORITY	Changes execution priority of a process or process pair
#ASSIGN (variable)	Holds information about all currently defined unit name
#BEGINTRANSACTION	Starts a new transaction
#CREATEPROCESSNAME	Creates unique process name
#CREATEREMOTENAME	Returns process name unique to specified system
#DEFINEADD	Adds a DEFINE to TACL context, using attributes in the working set
#DEFINEDELETE	Deletes a DEFINE from TACL context
#DEFINEDELETEALL	Deletes all DEFINEs from TACL context
#DEFINEINFO	Gets information about a DEFINE
#DEFINEMODE (variable)	Holds flag indicating whether DEFINEs can be used
#DEFINENAMES	Gets names of all DEFINEs that match specified template
#DEFINENEXTNAME	Gets name of next DEFINE following specified DEFINE in sequence established by the operating system
#DEFINEREADATTR	Gets value of specified attribute
#DEFINERESTORE	Creates or replaces active DEFINE, or replaces working set with contents of DEFINE previously saved with #DEFINESAVE
#DEFINERESTOREWORK	Restores DEFINE working set from background set
#DEFINESAVE	Saves copy of active DEFINE or working set for later restoration with #DEFINERESTORE
#DEFINESAVEWORK	Saves DEFINE current working set to background set
#DEFINESETATTR	Modifies value of specified DEFINE attribute in current working set

Table 7-6. Process Control Built-In Functions and Variables (page 2 of 3)

Function	Description
#DEFINESETLIKE	Initializes current working set with attributes of an existing DEFINE
#DEFINEVALIDATEWORK	Checks DEFINE current working set for consistency
#EMSADDSUBJECT	Adds subject token to event message buffer
#EMSADDSUBJECTV	Adds subject token to event message buffer, obtaining token values from a STRUCT
#EMSADDSUBJECTV	Adds subject token to event message buffer, obtaining token values from a STRUCT
#EMSGET	Retrieves token values from SPI buffer
#EMSGET	Retrieves token values from SPI buffer
#EMSGETV	Copies token values from SPI buffer to a STRUCT
#EMSINIT	Initializes a STRUCT as event message buffer
#EMSINITV	Initializes STRUCT as event message buffer, obtaining initial values from another STRUCT
#EMSTEXT	Converts information from event buffer to printable text
#EMSTEXTV	Converts information from event buffer to printable text, copies text to a STRUCT
#ENDTRANSACTION	Commits data base changes associated with a transaction
#HIGHPIN (variable)	Holds the default PIN range for processes started by the current TACL process
#INLINEECHO (variable)	Controls whether TACL echoes to its OUT file lines passed as input to inline processes
#INLINEEOF	Sends end-of-file to process running under control of INLINE facility
#INLINEOUT (variable)	Controls whether TACL copies to its own OUT file lines written by inline processes to their OUT files
#INLINEPREFIX (variable)	Holds prefix used to identify lines to be passed to inline processes instead of being acted upon by TACL
#INLINEPROCESS (variable)	Holds process ID of current inline process, if such exists
#INLINETO (variable)	Holds name of variable, if any, to which TACL appends lines written by inline processes to their OUT files
#LOOKUPPROCESS	Gets information about a PPD entry
#MOM	Obtains identity of creator process
#MYPID	Obtains your CPU,PIN number
#NEWPROCESS	Starts a process
#PARAM	Holds list of all your parameters, or a specified parameter
#PAUSE	Gives control of your terminal to another process

Table 7-6. Process Control Built-In Functions and Variables (page 3 of 3)

Function	Description
#PMSG (variable)	Holds state of PMSG flag
#PROCESS	Obtains identity of last process created or paused for by TACL
#PROCESSEXISTS	Determines whether a process exists
#PROCESSINFO	Requests information about a process
#SERVER	Creates and deletes servers

Managing the System Environment

Table 7-7 lists the commands that support system environment management.

Table 7-7. System Environment Management Commands (page 1 of 2)

Command	Description
ADDDSTTRANSITION	Adds entries to daylight-saving time transition table
ADDUSER (program)	Adds new users to a group
BACKUPCPU	Specifies backup CPU for current named TACL process, or deletes existing backup process
BUSCMD (program)	Tells operating system that a bus is (or is not) available for use
DEFAULT (program)	Changes logon default setting for volume and subvolume names; sets default disk file security
DELUSER (program)	Deletes users from a group
LOGOFF	Concludes TACL session
LOGON	Begins TACL session
O[BEY]	Instructs TACL to execute commands from file you specify
PASSWORD (program)	Establishes or changes your password
PMSEARCH	Defines subvolumes to be searched for program and macro files
RELOAD (program)	Reloads operating system image into a processor that was previously halted
REMOTEPASSWORD	Defines remote password for use in network security (runs RPASSWRD program)
RPASSWRD (program)	Establishes or changes remote password
SETPROMPT	Changes TACL prompt
SETTIME	Sets system date and time-of-day clocks
SINK	Disables the display of the result of the argument to SINK
SWITCH	Causes backup TACL process to become primary and primary process to become a backup
SYSTEM	Changes your current default system

Table 7-7. System Environment Management Commands (page 2 of 2)

Command	Description
VOLUME	Changes your current default volume, subvolume, or security
XBUSDOWN	Tells operating system that a bus is not available for use
XBUSUP	Tells operating system that a bus is available for use
YBUSDOWN	Tells operating system that a bus is not available for use
YBUSUP	Tells operating system that a bus is available for use

Table 7-8 lists the built-in functions and variables that support system environment management.

Table 7-8. System Environment Management Built-In Functions and Variables (page 1 of 2)

Function	Description
#ADDDSTTRANSITION	Adds entries to daylight-saving time transition table
#BACKUPCPU	Starts or stops the TACL backup process on a specified CPU
#BREAKMODE (variable)	Affects BREAK key operation
#CHANGEUSER	Logs user on under different user ID
#DEFAULTS (variable)	Holds volume or subvolume defaults you set
#DELAY	Causes TACL to wait for specified time
#INTERACTIVE	Determines whether your TACL is interactive
#LOGOFF	Logs off current TACL
#MYGMOM	Obtains identity of TACL job ancestor process
#MYSYSTEM	Determines name of system executing current TAC
#MYTERM (variable)	Holds name of your home terminal
#PMSEARCHLIST (variable)	Holds list of subvolumes to be searched for program and macro files
#PREFIX (variable)	Holds contents of prefix string
#PROMPT (variable)	Represents state of prompt flag
#REPLYPREFIX (variable)	Holds value of your reply prefix
#SETSYSTEMCLOCK	Changes setting of system clock
#SPIFORMATCLOSE	Closes an open EMS formatter template file
#SWITCH	Switches TACL to its backup process
#SYSTEM	Temporarily changes your default system
#SYSTEMNAME	Requests a system by name
#SYSTEMNUMBER	Requests a system by number
#TACLOPERATION	Determines whether TACL is reading commands from IN or $RECEIVE

Table 7-8. System Environment Management Built-In Functions and Variables (page 2 of 2)

Function	Description
#TACLSECURITY (variable)	Represents TACL security
#USERID	Specifies a user by user ID
#USERNAME	Specifies a user by user name
#XLOGON	Implements the LOGON command

Managing the TACL Environment

Table 7-9 lists the commands that support the TACL environment.

Table 7-9. TACL Environment Commands

Command	Description
ATTACHSEG	Provides user access to a TACL segment file
CREATESEG	Creates a TACL segment file
DETACHSEG	Relinquishes use of a TACL segment file
HOME	Specifies directory in which TACL searches first for variables
LOAD	Loads all properly formatted definitions from a TACL library
USE	Defines one or more directories in which TACL searches to find variables

Table 7-10 lists the built-in functions and variables that support the TACL environment.

Table 7-10. TACL Environment Commands

Function	Description
#ERRORNUMBERS	Holds information about latest error
#GETPROCESSSTATE	Obtains process state information about the current TACL process
#HOME	Represents your home directory
#LOAD	Processes a TACL library file
#SEGMENT	Obtains name of segment file that TACL is using for its variables
#SEGMENTCONVERT	Converts segment file from C00/C10 format to newer format
#SEGMENTINFO	Gets information about segments being used by TACL
#SETPROCESSSTATE	Sets process state flags for the current TACL process
#SETCONFIGURATION	Sets the flag settings that affect TACL behavior
#USELIST	Holds your use list

Processing Text in Variables

TACL provides two types of mechanisms that manipulate text in variables:

- Built-in functions and commands
- The #DELTA low-level text processor, accessed through the #DELTA built-in function

For more information about other functions and commands that process text, see the individual function and command descriptions in Section 8, UTILS:TACL Commands and Functions and Section 9, Built-In Functions and Variables. In addition, the *TACL Programming Guide* contains examples showing how to use these functions.

The commands in Table 7-11 provide data manipulation capabilities.

Table 7-11. Data Manipulation Commands (page 1 of 2)

Command	Description
_COMPAREV	Compares two variables
COMPUTE	Performs calculation and displays result
_CONTIME_TO_TEXT	Converts numeric date and time to text form
_CONTIME_TO_TEXT_DATE	Converts numeric date to text form
_CONTIME_TO_TEXT_TIME	Converts numeric time to text form
COPYVAR	Copies one variable to another
FILETOVAR	Copies data from a file and appends it to a variable
JOIN	Converts multiple-line variable to single-line variable
KEEP	Removes bottom levels from a variable stack
_LONGEST	Returns length of longest element in a space-separated list
_MONTH3	Returns three-letter abbreviation for a month number
OUTVAR	Displays contents of a variable
POP	Removes top level of variable or built-in function
PUSH A	Adds a level to a variable
SET VARIABLE	Changes contents of a variable level
VARTOFILE	Copies data from a variable to a file
VCHANGE	Changes all occurrences of one string to another string within a range of lines in a variable
VCOPY	Copies a range of lines from a variable and inserts it at a specified line position in another variable
VDELETE	Deletes a range of lines from a variable
VFIND	Searches a range of lines in a variable for a specified string

Table 7-11. Data Manipulation Commands (page 2 of 2)

Command	Description
VINSERT	Inserts lines from current TACL IN file at a specified line position
VLIST	Lists a range of lines in a variable
VMOVE	Deletes a range of lines from a variable, inserts them at a specified line position in another variable

Table 7-12 summarizes the built-in functions and variables that manipulate text in variables.

Table 7-12. Data Manipulation Built-In Functions and Variables (page 1 of 3)

Function	Description
#APPEND	Appends additional lines to a variable level
#APPENDV	Appends the contents of one variable level to the end of another variable level
#ARGUMENT	Parses arguments to routines
#CHARACTERRULES (variable)	Holds name of current character-processing rules file
#CHARADDR	Converts line address to character address
#CHARBREAK	Inserts line break in variable at character address
#CHARCOUNT	Obtains number of characters in variable
#CHARDEL	Deletes characters from variable at character address
#CHARFIND	Locates text in variable, searching forward from character address
#CHARFINDR	Locates text in variable, searching backward from character address
#CHARFINDRV	Locates string in variable, searching backward from character address
#CHARFINDV	Locates string in variable, searching forward from character address
#CHARGET	Obtains copy of specified number of characters from a variable
#CHARGETV	Copies specified number of characters from one variable to another
#CHARINS	Inserts text into a variable at character address
#CHARINSV	Inserts string into a variable at character address
#COMPAREV	Compares one variable with another
#COMPUTE	Returns value of expression
#COMPUTEJULIANDAYNO	Converts Gregorian calendar date to a Julian day number

Table 7-12. Data Manipulation Built-In Functions and Variables (page 2 of 3)

Function	Description
#COMPUTETIMESTAMP	Converts calendar date to a four-word timestamp
#COMPUTETRANSID	Converts separate components of a transaction ID to one numeric transaction ID
#CONTIME	Converts timestamp to seven-digit date and time
#CONVERTPHANDLE	Converts process file identifier to process handle or vice versa
#CONVERTPROCESSTIME	Converts time value obtained by PROCESSTIME option of #PROCESSINFO
#CONVERTTIMESTAMP	Converts GMT timestamp to a local-time-based timestamp, or local-time-based timestamp to a GMT timestamp
#DEF	Defines a variable
#DELTA	Acts as a complex character processor
#EMPTY	Determines whether specified string contains text
#EMPTYV	Determines whether a variable level contains any lines
#EXTRACT	Deletes first line of a variable level
#EXTRACTV	Moves first line of a variable level to another variable
#FRAME	Tracks pushed variables
#GETSCAN	Obtains number of characters passed over by #ARGUMENT
#INFORMAT (variable)	Represents formatting mode for #INPUT
#INTERPRETJULIANDAYNO	Converts Julian day number to year, month, and day
#INTERPRETTIMESTAMP	Breaks down four-word timestamp to its component parts
#INTERPRETTRANSID	Converts numeric transaction ID to its separate component values
#JULIANTIMESTAMP	Obtains four-word timestamp
#KEEP	Removes all but specified level of a variable
#LINEADDR	Converts character address to line address
#LINEBREAK	Inserts line break in variable at line address
#LINECOUNT	Obtains number of lines in a variable
#LINEDEL	Deletes lines from variable at line address
#LINEFIND	Locates text in variable, searching forward from line address
#LINEFINDR	Locates text in variable, searching backward from line address
#LINEFINDRV	Locates string in variable, searching backward from line address

Table 7-12. Data Manipulation Built-In Functions and Variables (page 3 of 3)

Function	Description
#LINEFINDV	Locates string in variable, searching forward from line address
#LINEGET	Gets copy of specified number of lines from a variable
#LINEGETV	Copies specified number of lines from one variable to another
#LINEINS	Inserts text into a variable at line address
#LINEINSV	Inserts string into a variable at line address
#LINEJOIN	Deletes line break at end of a line, joining following line to it

Controlling Program Flow

Table 7-13 lists built-in functions and variables that provide program control flow for TACL programs.

Table 7-13. Flow Control Built-In Functions and Variables

Function	Description
#CASE	Chooses one out of a set of options
#ERRORTEXT	Used with exception handlers to catch error text
#EXCEPTION	Determines why a routine was invoked during exception handling
#EXIT	Holds state of exit flag
#FILTER	Indicates which exceptions a routine can handle
#IF	Executes one of two options
#LOOP	Repeatedly executes one or more statements in a function
#RAISE	Defines exception to be filtered by routines
#RETURN	Exits from a routine immediately
#ROUTINENAME	Obtains name of variable in which containing routine resides
#WAIT	Specifies variables for which a routine must wait

Debugging TACL Statements

Table 7-14 lists the commands that provide debugging support.

Table 7-14. Debugging Commands

Command	Description
BREAK	Sets breakpoint on specified variable or lists all breakpoints
DEBUG	Puts a process into debug state
_DEBUGGER	Debugs TACL statements
SET INSPECT	Specifies whether INSPECT or DEBUG is default debugger for programs started by TACL

Table 7-15 lists built-in functions and variables that provide debugging support.

Table 7-15. Debugging Built-In Functions and Variables

Function	Description
#BREAKPOINT	Sets or deletes _DEBUGGER breakpoint for a specific variable level
#DEBUGPROCESS	Calls debugger for specified process
#INSPECT	Holds state of INSPECT flag
#TRACE	Represents state of TRACE flag

8

UTILS:TACL Commands and Functions

This section describes the TACL commands and functions that are located in the :UTILS:TACL directory. These commands and functions are typically used for interactive tasks. Each description contains:

- A summary of the action of the command, function, or program
- The syntax of the command, function, or program, including a description of the syntax of parameters
- The listing format used for output (if the command, function, or program produces a display or listing output)
- Considerations for the use of the command, function, or program
- Examples of use of the command, function, or program

Note. All examples in this section are based on the assumptions that the built-in variable #INFORMAT has been set to TACL, which enables recognition and processing of the TACL special characters; that the built-in variable #PMSEARCHLIST has been set to include $SYSTEM.SYSTEM, and the keyword #DEFAULTS, which enables the use of implied RUN commands; and that the required TACL library files have been loaded into memory.

:UTILS:TACL Command Summary

These tables summarize the :UTILS:TACL commands and functions and the utility programs that are also available.

Commands and Programs

The commands and functions listed in Table 8-1 are available to all users.

Table 8-1. Commands and Programs (page 1 of 5)

Command or Function	Description
ACTIVATE Command	Reactivates a previously suspended process
ADD DEFINE Command	Creates one or more DEFINEs
ALTER DEFINE Command	Changes attributes of one or more DEFINEs
ALTPRI Command	Alters execution priority of a process
ASSIGN Command	Gives file attributes to logical-file descriptions used by application programs
ATTACHSEG Command	Provides user access to TACL segment file

Table 8-1. Commands and Programs (page 2 of 5)

Command or Function	Description
BACKUPCPU Command	Specifies backup CPU for current named TACL process, or deletes existing backup process
BREAK Command	Sets breakpoint on specified variable or lists all breakpoints
BUILTINS Command	Shows TACL built-in functions and variables
CLEAR Command	Deletes attributes previously set by ASSIGN or PARAM commands
CLICVAL Program	Validates the correctness and applicability of the Core License file for the system.
COLUMNIZE Command	Displays a list in columnar form
COMMENT Command	Begins comment line in TACL command file
_COMPAREV Function	Compares two variables
COMPUTE Command	Performs calculation and displays result
_CONTIME_TO_TEXT Function	Converts numeric date and time to text form
_CONTIME_TO_TEXT_DATE Function	Converts numeric date to text form
_CONTIME_TO_TEXT_TIME Function	Converts numeric time to text form
COPYDUMP Program	Copies and compresses tape dump file or existing disk dump file into a disk dump file. Not supported on H-series systems.
COPYVAR Command	Copies one variable to another
CREATE Command	Creates an unstructured disk file
CREATESEG Command	Creates a TACL segment file
DEBUG Command	Puts a process into debug state
DEBUGGER Function	Debugs TACL statements
DEFAULT Program	Changes logon default setting for volume or subvolume names; sets default disk file security
DELETE DEFINE Command	Removes one or more DEFINEs from
DETACHSEG Command	Relinquishes use of a TACL segment file
ENV Command	Displays settings of TACL environmental parameters
EXIT Command	Used interactively, stops current process; used in a command file, stops execution of commands
FC Command	Retrieves, edits, and reexecutes lines in history buffer
FILEINFO Command	Displays information about files
FILENAMES Command	Displays names of files in a subvolume, using a file-name template

Table 8-1. Commands and Programs (page 3 of 5)

Command or Function	Description
FILES Command	Displays names of files in a subvolume
FILETOVAR Command	Copies data from a file and appends it to a variable
HELP Command	Displays useful information about TACL
HISTORY Command	Displays previously issued command lines
HOME Command	Specifies directory in which TACL searches first for variables
INFO DEFINE Command	Displays attributes and associated values in one or more DEFINEs residing in process file segment of current TACL process
INITTERM Command	Initializes home terminal by reinstating its default SETMODE settings
INLECHO Command	Controls copying to TACL OUT file of lines sent to inline process
INLEOF Command	Sends end-of-file indication to an inline process
INLOUT Command	Controls copying to TACL OUT file of lines sent to OUT file of inline process
INLPREFIX Command	Establishes prefix that identifies lines to be passed to inline process
INLTO Command	Establishes variable to receive copies of lines sent to OUT file of inline process
IPUCOM Program	Displays, sets, or resets an IPU number associated with a process. Also used to set or display CPU-wide controls.
JOIN Command	Converts multiple-line variable to single-line variable
KEEP Command	Removes bottom levels from a variable stack
KEYS Command	Displays current function-key variable
LOAD Command	Loads all properly formatted definitions from a TACL library
LOADEDFILES Command	Displays all files loaded by a selected process
LOGOFF Command	Concludes TACL session
LOGON Command	Begins TACL session
_LONGEST Function	Returns length of longest element in a space-separated list
_MONTH3 Function	Returns three-letter abbreviation for a month number
O[BEY] Command	Instructs TACL to execute commands from file you specify
OUTVAR Command	Displays contents of a variable

Table 8-1. Commands and Programs (page 4 of 5)

Command or Function	Description
PARAM Command	Assigns parameter value to a parameter name, or displays all current parameters and their values
PASSWORD Program	Establishes or changes your password
PAUSE Command	Makes TACL stop prompting for commands and allows another process to control terminal
PMSEARCH Command	Defines subvolumes to be searched for program and macro files
PMSG Command	Controls process-identifier displays of processes you create or delete
POP Command	Removes top level of variable or built-in function
POSTDUMP Utility	Enables explicit postdump processing of an existing full dump file to produce an extracted dump file.
PPD Command	Displays names, *cpu,pin* designations, and ancestors of processes currently running
PURGE Command	Purges (deletes) a disk file
PUSH Command	Adds a level to a variable
REMOTEPASSWORD Command and RPASSWRD Program	Defines remote password for use in network security (runs RPASSWRD program)
RENAME Command	Renames a disk file
RESET DEFINE Command	Restores attributes in DEFINE working set to their initial values
REMOTEPASSWORD Command and RPASSWRD Program	Establishes or changes remote password
RUN[D|V] Command	Runs a program; optionally puts resulting process into debug state
SEGINFO Command	Shows information about TACL segment files
SEMSTAT Program	Prints BINSEM usage information and statistics for a process whose ID or process name is provided
SET DEFINE Command	Establishes a value for one or more DEFINE attributes in working attribute set
SET DEFMODE Command	Controls whether DEFINEs are enabled for current TACL process and are propagated to new processes
SET HIGHPIN Command	Specifies the default PIN range for processes started by the TACL process
SET INSPECT Command	Specifies whether INSPECT or DEBUG is default for programs started by TACL

Table 8-1. Commands and Programs (page 5 of 5)

Command or Function	Description
SET SWAP Command	Sets swap volume for all subsequent RUN commands (unless swap volume is explicitly specified in a command)
SET VARIABLE Command	Changes contents of a variable level
SETPROMPT Command	Changes TACL prompt
SHOW Command	Displays values of attributes set with SET command
SHOW DEFINE Command	Displays DEFINE attributes in working attribute set and their current values
SINK Command	Disables the display of the result of its argument
STATUS Command	Displays status of running processes
STOP Command	Stops and deletes a process
SUSPEND Command	Prevents a process from running until it is reactivated by an ACTIVATE command
SWITCH Command	Causes backup TACL process to become primary, and primary process to become a backup
SYSTEM Command	Changes your current default system
SYSTIMES Command	Displays current date and time, plus date and time of last cold load
TACL Program	Starts TACL process on your local system or a remote system
TIME Command	Displays current system date and time of day
USE Command	Defines one or more directories in which TACL searches to find variables
USERS Program	Displays attributes of users and groups
VARIABLES Command	Displays names of all your variables
VARINFO Command	Displays information about specified variable
VARTOFILE Command	Copies data from a variable to a file
VCHANGE Command	Changes all occurrences of one string to another string within a range of lines in a variable

Restricted Commands

Only group managers (user ID *n*, 255) can execute the commands in Table 8-2.

Table 8-2. Group Manager Commands

Command or Function	Description
ADDUSER Program (Group Managers Only)	Adds new users to a group
DELUSER Program (Group Managers Only)	Deletes users from a group

Only super-group users (user ID 255,*n*) can execute the commands in Table 8-3.

Table 8-3. Super-Group User Commands

Command or Function	Description
ADDDSTTRANSITION Command (Super-Group Only)	Adds entries to daylight savings time transition table
ALARMOFF Program (Super-Group Only)	Turns off NonStop VLX system audio alarm
BUSCMD Program (Super-Group Only)	Alerts operating system that a bus is (or is not) available for use
LIGHTS Program (Super-Group Only)	Controls processor panel lights
RCVDUMP Program (Super-Group or Super ID Only)	Receives a dump from a halted processor over an interprocessor bus
RECEIVEDUMP Command (Super-Group Only)	Receives a dump from a halted processor over an interprocessor bus (runs RCVDUMP). Not supported on H-series systems.
RELOAD Program (Super-Group Only)	Reloads operating system image into a processor that was previously halted
SETTIME Command (Super-Group Only)	Sets system date and time-of-day clocks
XBUSDOWN/YBUSDOWN Command (Super-Group Only)	Alerts operating system that a bus is not available for use
XBUSUP/YBUSUP Command (Super-Group Only)	Alerts operating system that a bus is available for use

:UTILS:TACL Command Descriptions

The remainder of this section contains descriptions of the syntax for the commands and functions in :UTILS:TACL and the programs used by TACL.

ACTIVATE Command

Use the ACTIVATE command to restart a process previously suspended by the SUSPEND Command on page 8-217 or the #SUSPENDPROCESS Built-In Function on page 9-394.

```
ACTIVATE [ [\node-name.]{$process-name | cpu,pin } ]
```

\node-name

is the system where the process resides.

$process-name

is the name of the process or process pair you want to restart.

cpu,pin

is the CPU number and process identification number for the process you want to restart.

Considerations

- If you omit the process specification, ACTIVATE restarts the last process TACL started or for which TACL paused, if that process is still running. That process is called the default process. You can use the #PROCESS Built-In Function on page 9-290 to determine the name or CPU and PIN of the default process. If no default process exists, you must include a process specification.
- The super ID can activate any suspended process.
- A group manager whose process accessor ID matches the user ID of any member of the group can activate any suspended process owned by someone in the group. (For a discussion of process accessor IDs, see the *Guardian User's Guide*. For restrictions that apply to processes running on remote systems, see the *Expand Network Management and Troubleshooting Guide*.)
- Users other than super-group users and group managers can activate only those processes with a process accessor ID that matches their own user ID.
- A restarted process is placed in the queue of processes that are waiting for execution. In the queue, the process with the highest priority is executed first, so a newly activated process immediately begins execution only if it has the highest execution priority. You can set execution priority with the ALTPRI Command on page 8-20.

ADD DEFINE Command

Use the ADD DEFINE command to create one or more DEFINEs in the process file segment (PFS) of the current TACL. For a description of DEFINEs, see Section 5, Statements and Programs.

```
ADD DEFINE {define-name}
           {define-name [, define-name ] ... )}
  [ , LIKE define-name ] [ , attribute-spec ] ...
```

define-name

is the name to be assigned to the DEFINE created by this command. A DEFINE name can be from 2 to 24 characters in length; the first character must be an equal sign (=) and the second must be a letter. The name can contain alphanumeric characters, hyphens (-), underscores (_), or circumflexes (^).

If you specify neither LIKE clause nor *attribute-spec*, you create one or more DEFINEs with the attributes and values of the working attribute set.

LIKE *define-name*

creates a DEFINE identical to define-name but modified by attribute-spec (if present). *define-name* is the name of an existing DEFINE.

attribute-spec

is one or more clauses where each clause specifies the name of a valid DEFINE attribute and the value you want to associate with it in the new DEFINE. If you include a LIKE clause, TACL creates a DEFINE with the attributes and values of that *define-name*, modified by the content of *attribute-spec*. If you do not include a LIKE clause, TACL creates a DEFINE with the attributes and values of the working attribute set, modified by the content of *attribute-spec*.

See the SET DEFINE Command on page 8-173 for descriptions of valid DEFINE attributes.

Considerations

- To modify the working attribute set before you create a DEFINE, use the SET DEFINE command. To display the working attribute set or the attributes that are currently set or defaulted, use the SHOW DEFINE command.
- Attributes you set in an ADD DEFINE command (known as the ADD DEFINE attribute set) do not become part of the working attribute set.
- The ADD DEFINE command checks for consistency among the current attributes (this includes attributes you set before entering the ADD DEFINE command, as well as the ADD DEFINE attribute set). If the current attributes are incomplete or inconsistent, an error occurs and no DEFINE is created. For example, if you enter

an ADD DEFINE command that does not make a complete attribute set (that is, if a required attribute is still missing), TACL displays this message:

```
Current attribute set is incomplete
```

- If you include a LIKE clause in an ADD DEFINE command, it is processed first, before any *attribute-spec* clauses. Then each *attribute-spec* clause is processed in order. The LIKE clause establishes a set of DEFINE attributes for the new DEFINE; those attributes are then modified by the *attribute-spec* clauses, if there are any. If you include a LIKE clause, it must precede any *attribute-spec* clauses in the ADD DEFINE command.

  If you do not include a LIKE clause, any *attribute-spec* clauses in your ADD DEFINE command are processed in order.

- Because the CLASS attribute works like a subtype of DEFINE, CLASS affects the ADD DEFINE command in these ways:

  - Including CLASS in an ADD DEFINE command clears all existing attribute settings. If you use the CLASS attribute, make it the first attribute in your ADD DEFINE command or include it in your first SET DEFINE command.
  - To avoid errors or unexpected results, do not specify the CLASS attribute in a command that includes a LIKE clause.
  - You cannot specify an attribute that is not valid for the CLASS of the DEFINE you are adding. For example, when the default CLASS is in effect (CLASS MAP), entering this command produces this error message:

    ```
    79> ADD DEFINE =TAPE1, LABELS IBM
    There is no attribute "LABELS" for the current class
    ```

- When a backup TACL process takes over, TACL deletes existing DEFINEs.
- To obtain error information, use the #ERRORNUMBERS Built-In Variable on page 9-160.

Examples

1. Suppose that you are using a long file name in a series of commands or procedure calls. Using a MAP DEFINE for file-name redirection, you can substitute a shorter name for the long name. For example, this command sets up a MAP DEFINE (by default) with the name =PLUTO:

   ```
   80> ADD DEFINE =PLUTO, FILE \FAR.$OFF.WORLDS.PLUTO
   ```

   Now you can use the name =PLUTO wherever you would have used the longer file name, and the system knows that you mean \FAR.$OFF.WORLDS.PLUTO.

2. This command sets up a TAPE DEFINE named =S2 that describes a tape file on the IBM standard labeled tape volume number 58. If you specify LABELS IBM, you must also specify FILEID. Because the FILESEQ attribute defaults to 1, the file is

the first one on the tape. Data is to be translated from the ASCII format to EBCDIC. The USE EXTEND attribute indicates that data is to be added to the end of the file.

```
81> ADD DEFINE =S2, CLASS TAPE, LABELS IBM, FILEID $TAPE,&
81> &VOLUME 58, EBCDIC OUT, USE EXTEND
```

ADDDSTTRANSITION Command (Super-Group Only)

Use the ADDDSTTRANSITION command to add entries to the daylight savings time (DST) transition table.

```
ADDDSTTRANSITION start-date-time , stop-date-time , offset
```

start-date-time

is the beginning of the period where offset is applicable. The format is:

```
{ month day | day month } year,hour[:sec] GMT | LST
```

where *month* is the first three letters of the name of the month; GMT indicates Greenwich mean time; and LST indicates local standard time.

stop-date-time

is the end of the period where offset is applicable. The format is:

```
{ month day | day month } year,hour[:sec] GMT | LST
```

where *month* is the first three letters of the name of the month; GMT indicates Greenwich mean time; and LST indicates local standard time.

offset

is the difference between standard time and daylight-saving time. Specify *offset* as:

```
[ + | - ] hour: min
```

The offset must be between -8:59 and +8:59.

Considerations

- To use the ADDDSTTRANSITION command, you must have a super-group ID (255,*n*).
- The ADDDSTTRANSITION command can be used only in systems for which the TABLE option of the DAYLIGHT_SAVINGS_TIME clause was specified when SYSGEN was run to create the current system image.
- The table of daylight savings time transitions must be initialized with at least one DST transition that is earlier than the current date and time and with at least two transitions that are later than the current date and time.
- 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.
- You can include ADDDSTTRANSITION commands in a command file that is invoked as an IN file for the initial TACL process in a system.

Example

To add two periods of daylight savings time (from April 1, 1986, to September 1, 1986, and from April 1, 1986, to September 1, 1987), enter:

```
69> ADDDSTTRANSITION 01 APR 1986, 2:00 LST, 01 SEP 1986, &
69> &2:00 LST, 1:00

70> ADDDSTTRANSITION 01 SEP 1986, 2:00 LST, 01 APR 1987, &
70> &2:00 LST, 0:00

71> ADDDSTTRANSITION 01 APR 1987, 2:00 LST, 01 SEP 1987, &
71> &2:00 LST, 1:00
```

ADDUSER Program (Group Managers Only)

Run the ADDUSER program to add new group IDs and user IDs to the system. To use the ADDUSER program, you must have a group-manager ID (*group,*255)or the super ID (255,255).

```
ADDUSER [ / run-option [ , run-option ] ... / ]
  group-name.user-name , group-id, user-id
```

run-option

is any of the options described in the RUN[D|V] Command on page 8-156.

group-name.user-name

are the group and individual names, respectively, of the new user. Each name can contain from one to eight letters or digits, and the first character must be a letter.

group-id

is an integer in the range from 0 through 255 that uniquely identifies a group. 255 is reserved as the super-group ID.

user-id

is an integer in the range from 0 through 255 that uniquely identifies a user within a group. 255 is reserved for group managers (*group*,255) and the super ID (255,255).

Considerations

- Only a group manager or the super ID can add new users to a system. Group managers can add new users to their respective groups; the super ID can add new users to any group.
- The super ID can create a new group by adding a new user with a previously unused *group-id* and *user-id*.
- A group does not need to have a group manager.
- The logon defaults for a new user who was just added to the system are volume $SYSTEM, subvolume NOSUBVOL, and disk file security “AAAA”.

Examples

1. Assume that there is no group name MANUF and no group ID 8 in the system. In one ADDUSER command, the super ID can create a new group (MANUF) and add a new user (STELLA) with group ID 8 and user ID 1:

```
12> ADDUSER MANUF.STELLA, 8,1
MANUF.STELLA (8,1) HAS BEEN ADDED TO THE USERID FILE.
```

2. The super ID can also add a manager to the MANUF group by adding a user with user ID 255 and group ID 8:

```
13> ADDUSER MANUF.HONCHO, 8,255
MANUF.HONCHO (8,255) HAS BEEN ADDED TO THE USERID FILE.
```

3. The new manager can now add other users to the group:

```
14> ADDUSER MANUF.MABEL, 8,2
MANUF.MABEL (8,2) HAS BEEN ADDED TO THE USERID FILE.
15> ADDUSER MANUF.FRED, 8,44
MANUF.FRED (8,44) HAS BEEN ADDED TO THE USERID FILE.
```

ALARMOFF Program (Super-Group Only)

Run the ALARMOFF program to turn off the NonStop VLX system audio alarm. If you issue this command for systems other than a NonStop VLX, TACL returns the error message:

```
THIS SYSTEM DOES NOT HAVE AN ALARM.
```

You must use a super-group user ID (255,*user-id*) to issue this command.

```
ALARMOFF
```

ALTER DEFINE Command

Use the ALTER DEFINE command to change the attributes of one or more existing DEFINEs in the process file segment (PFS) of the current TACL. For a description of DEFINEs, see Section 5, Statements and Programs.

```
ALTER DEFINE define-name-list {, attribute-spec}
                              {, RESET reset-list}
```

define-name-list

is a list of one or more existing DEFINEs whose attributes you want to alter. For *define-name-list*, specify either of these:

define-template
(*define-template* [, *define-template*] ...)

define-template

is a DEFINE name that can optionally contain template characters:

- * matches zero or more characters
- ? matches a single character

A DEFINE template that consists entirely of =* or ** causes all existing DEFINEs to be displayed.

attribute-spec

is a sequence of one or more DEFINE attributes and the value or values you want to associate with each attribute. You can specify new values for attributes that are already established in a DEFINE specified in *define-name-list*, and you can specify values for new attributes, ones that are not established in the *define-name-list*. The syntax of *attribute-spec* is described in the SET DEFINE Command on page 8-173.

RESET *reset-list*

restores the values of one or more attributes associated with the DEFINEs in *define-name-list* to their initial settings. For *reset-list*, specify one or more DEFINE attributes, in either of these forms:

attribute-name
(*attribute-name* [, *attribute-name*]...)

reset-list cannot include any required attribute; see the discussion of attribute names and values for the SET DEFINE Command on page 8-173.

Considerations

- An ALTER DEFINE command affects existing DEFINE statements only and does not change the working attribute set. Similarly, an ALTER DEFINE command affects DEFINEs for the current TACL process only; any DEFINE that was propagated from the current TACL to another process is unchanged.
- Because the CLASS attribute establishes new attributes for a DEFINE, you should keep these points in mind when you use the ALTER DEFINE command:
  - Attributes are altered in the order in which they are specified in the ALTER DEFINE command.
  - If you include the CLASS attribute, all new attributes are established for the DEFINE; any existing attributes are erased (including any attributes preceding CLASS in the ALTER command itself). The new attributes are those associated with the specified CLASS, and each attribute has its initial setting.
  - You cannot alter an attribute that is not valid for the class of that DEFINE. For example, if =DFILE is CLASS TAPE, this command produces this error:

```
32> ALTER DEFINE =DFILE, FILE $MUNCH.NUMBERS.DIGIT
There is no attribute "FILE" for the current class
```

- Before a DEFINE is altered, the ALTER DEFINE command checks for consistency among the new attribute values specified and the other existing attributes of the DEFINE. If the attributes are incomplete or inconsistent, an error occurs and no changes are made to the DEFINE.
- When a backup TACL process takes over, TACL deletes existing DEFINEs.
- To obtain error information, use the #ERRORNUMBERS Built-In Variable on page 9-160.

Example

This ALTER DEFINE command changes the DEVICE attribute in the DEFINEs named =ONE and =TWO so that when the DEFINE is opened, the tape process searches for the tape files MAYRCDS and JUNRCDS on the tape mounted on tape drive $TAPE1:

```
27> INFO DEFINE (=ONE, =TWO), DETAIL
DEFINE NAME=ONE
CLASS TAPEVOLUME 4335
LABELS ANSI
FILEID MAYRCDS
DEVICE $TAPE

DEFINE NAME =TWO
CLASS TAPE
VOLUME 4335
LABELS ANSI
```

```
FILEID JUNRCDS
DEVICE $TAPE

28> ALTER DEFINE ( =ONE, =TWO ), DEVICE $TAPE1
29> INFO DEFINE (=ONE, =TWO), DETAIL
DEFINE NAME =ONE
CLASS TAPE
VOLUME 4335
LABELS ANSI
FILEID MAYRCDS
DEVICE $TAPE1

DEFINE NAME =TWO
CLASS TAPE
VOLUME 4335
LABELS ANSI
FILEID JUNRCDS
DEVICE $TAPE1
```

ALTPRI Command

Use the ALTPRI command to change the execution priority of a process or process pair.

```
ALTPRI [\node-name.]{$process-name | cpu,pin } , pri
```

\node-name

is the system where the process resides.

$process-name

is the name of the process or process pair.

cpu,pin

is the CPU number and process identification number for the process.

pri

is the new execution priority of the process. It is an integer in the range from 1 through 199 (1 is lowest priority).

Considerations

If you do not specify a process, ALTPRI changes the priority of the default process. The default process is the process that was last started by the current TACL or for which TACL most recently paused, if that process still exists. To determine the default process, use the #PROCESS Built-In Function on page 9-290.

The super ID can change the priority of any process in the system.

A group manager can alter the priority of any process whose process accessor ID matches any user ID in the group.

Users other than group managers can change the priority of only those processes whose process accessor IDs match their user ID. (For a description of process accessor IDs and creator accessor IDs, see the *Guardian User's Guide*.)

Before increasing the priority of a process, carefully consider the effect the change might have on system performance. For example, assigning a high priority to processes with a large amount of CPU activity, such as those involving lengthy arithmetic computations, can significantly degrade system performance.

Example

Assume that a process named $SLOW is currently running with an execution priority of 110. You can raise the execution priority of $SLOW to 140 by entering:

```
12> ALTPRI $SLOW, 140
```

ASSIGN Command

Use the ASSIGN command to assign names of actual files to logical file names used in programs (such as those written in COBOL, FORTRAN, and other programming languages) and, optionally, to specify the characteristics of such files. If you omit the parameters of the ASSIGN command, ASSIGN displays the assigned values for all assignments currently in effect.

```
ASSIGN [ logical-unit [ , [ actual-file-name ]
  [ , create-open-spec ] ... ] ]
```

logical-unit

is the name to which a file name or file attributes are assigned. For *logical-unit*, specify one of these:

```
*. logical-file
program-unit.logical-file
logical-file
```

Both *program-unit* and *logical-file* names consist of 1 to 31 alphanumeric, hyphen (-), or circumflex (^) characters.

The exact meanings of program-unit, the asterisk (*), and logical-file depend on the application; in general:

- *program-unit* is the name used in the source program to which the file-name assignment is to apply.
- * applies the assignment to all program units in the object program file being run.
- *logical-file* is the name of the file as given in the source program.

For details on treatment of ASSIGN and PARAM commands by other programming languages, see the appropriate language manual.

actual-file-name

is the name of the actual physical file. A partial file name is not expanded; however, the application process can expand the file name using the default information passed in the startup message.

If you omit both *actual-file-name* and *create-open-spec*, the current assignment value for logical-unit is displayed.

If you omit *actual-file-name* but include *create-open-spec*, spaces are passed in the *actual-file-name* field of the assign message.

create-open-spec

is a file-creation or file-open specification that sets certain file attributes. For *create-open-spec*, specify one of these:

```
extent-spec
exclusion-spec
access-spec
CODE file-code
REC record-size
BLOCK block-size
```

extent-spec

is the size of the file extents allocated to the file. Specify either of these:

```
EXT [ ( ] pri-extent-size [ )
EXT ( [ pri-extent-size ] , sec-extent-size )
```

pri-extent-size

is the size of the primary file extent to be allocated to the file. It is an integer in the range from 1 through 65535. (For treatment of *pri-extent-size* by applications written in other languages, see the appropriate language manual.)

sec-extent-size

is the size of the secondary extents, allocated to the file after the primary extent is allocated. It is an integer in the range from 1 through 65535. (For details on treatment of *sec-extent-size* by applications written in other languages, see the appropriate language manual.)

exclusion-spec

is the exclusion mode for logical-unit. It determines the circumstances under which other processes can access the file. Specify *exclusion-spec* as one of these:

```
EXCLUSIVE
SHARED
PROTECTED
```

EXCLUSIVE

means that no other processes can access *actual-file-name* while the program that refers to *logical-unit* has the file open.

SHARED

means that other processes can both read and write to *actual-file-name* while the program that refers to *logical-unit* has the file open.

PROTECTED

means that another process can read, but not write to, *actual-file-name* while the program that refers to *logical-unit* has the file open.

For more information about exclusion modes, see the *ENSCRIBE Programmer's Guide*.

access-spec

is the access mode for *logical-unit*. It specifies the file operations that can be performed. Specify *access-spec* as one of these:

```
I-O
INPUT
OUTPUT
```

I-O

processes can both read the file and write to it.

INPUT

processes can only write to the file.

OUTPUT

processes can only read the file.

For more information about access modes, see the *ENSCRIBE Programmer's Guide*.

CODE *file-code*

assigns a file code to *logical-unit*. Specify file-code as an integer in the range from 0 through 65535. If *file-code* is omitted, the code is set to 0. (See the *File Utility Program (FUP) Reference Manual* for a table of reserved codes.)

REC *record-size*

sets the length of records in logical-unit. Specify *record-size* as an integer in the range from 1 through 65535. (For details on FORTRAN or COBOL application treatment of REC *record-size*, see the appropriate language manual.)

BLOCK *block-size*

sets the size of the data blocks used by logical-unit. Specify *block-size* as an integer in the range from 1 through 65535. (For details on FORTRAN or COBOL application treatment of REC BLOCK record-size, see the appropriate language manual.)

Considerations

- Use the CLEAR command to delete existing ASSIGNs.
- The ASSIGN command only stores the values it assigns and sends those values to requesting processes in the form of assign messages; it neither files nor interprets the assigned values. Those tasks must be done by the application program.
- TACL creates an assign message for each ASSIGN command in effect. A new process must request its assign messages (if any) following receipt of the startup message. The COBOL and FORTRAN compilers provide the code for this function. TAL programs that use ASSIGN commands must provide their own code for handling assign messages.
- The LOGOFF command deletes existing assignments. If you enter a second LOGON command without first logging off, however, all assignments are retained, even if your user ID is changed.
- If you start a new TACL process from your existing TACL process, the new TACL process does not inherit existing ASSIGN values.
- When a backup TACL process takes over, TACL deletes existing assignments.
- From a TACL macro or routine, you can use the #ASSIGN built-in function to associate an actual file name with a logical file name.
- The same set of ASSIGN attributes can be configured for a generic process through SCF. For the syntax, see the *SCF Reference Manual for the Kernel Subsystem*.

Examples

1. You can assign an actual file name and file-creation attributes to the logical file PRTFILE used by a COBOL program by entering:

   ```
   14> ASSIGN prtfile, myfile, EXT 4096, CODE 9999,&
   14> &EXCLUSIVE, OUTPUT
   ```

2. You can assign an actual file name and file-creation attributes to the logical file FT002 used by a FORTRAN program by entering:

   ```
   15> ASSIGN FT002, datafile, INPUT, EXCLUSIVE
   ```

3. You can get a list of the attributes of the logical file PRTFILE by entering:

   ```
   16> ASSIGN prtfile
   ```

   Here is an example of the information displayed:

   ```
   PRTFILE

            Physical file:    MYFILE
            Primary extent:   4096
            File code:        9999
   ```

```
          Exclusion:        EXCLUSIVE
          Access:          OUTPUT
```

4. You can list the assigned attributes of all logical files by entering:

   ```
   17> ASSIGN
   ```

   Here is an example of the information displayed:

   ```
   PRTFILE

             Physical file:    MYFILE
             Primary extent:   4096
             File code:        9999
             Exclusion:        EXCLUSIVE
             Access:          OUTPUT

   FT002

             Physical file:    DATAFILE
             Exclusion:        EXCLUSIVE
             Access:          INPUT
   ```

ATTACHSEG Command

Use the ATTACHSEG command to load an existing TACL segment file into memory and associate it with a directory so that TACL can gain quick access to its variables.

```
ATTACHSEG {PRIVATE | SHARED} file-name directory-name
```

PRIVATE

specifies that the segment is to be opened for use by only one process (yours) and that data can be written into it.

SHARED

specifies that the segment is to be opened in the read-only mode and that other processes can open the segment for reading only.

file-name

is the name of a TACL segment file, created previously by a CREATESEG command.

directory-name

is the name of a variable that is to be pushed and set to a directory of the variables in the segment file.

Considerations

- You cannot attach a segment that resides on another system.
- You cannot attach more than 50 segment files.
- TACL segment files have file code 440.
- If access is PRIVATE:
  - You must have both read and write access to the segment file.
  - There is a delay while TACL reads the segment file. A segment file typically contains unused space; TACL reads only the valid data within the file, not the entire file.
  - The segment file remains open and unchanged until you issue a DETACHSEG command for it. If for any reason TACL stops while updating the file during the DETACHSEG operation, the file contents are unpredictable, and any future attempts to attach the file will fail.
- If access is SHARED, you must have at least read access to the segment file.
- To display a table of information about all the segment files in use by your TACL process, use the SEGINFO Command on page 8-168.

- For more information about segment files, see the CREATESEG Command on page 8-46.

Note. The PRIVATE option allows access by only one process. You cannot access a segment file if you have opened it with PRIVATE access with another TACL process.

Examples

This command loads the segment file MYSEGFIL into memory and records the names of the loaded variables in the directory MYDIR. Other users can access the segment as well.

```
23> ATTACHSEG SHARED mysegfil :mydir
```

These commands attach two segment files to a directory. The second file is attached through a subdirectory:

```
23>ATTACHSEG PRIVATE myseg :mydir
24>ATTACHSEG SHARED yourseg :mydir:subdir
```

BACKUPCPU Command

Use the BACKUPCPU command to start a backup process for the current TACL process or to replace an existing backup process. The BACKUPCPU command is an alias for the #BACKUPCPU built-in function.

```
BACKUPCPU [ cpu ]
```

cpu

is the number of the processor where the backup TACL process is to be started. Specify *cpu* as an integer in the range from 0 through 15. If you omit *cpu*, BACKUPCPU deletes the existing backup process.

Considerations

- BACKUPCPU with no *cpu* specification has no effect if the current TACL process has no backup.
- Only a named TACL process can have a backup. (For more information on named processes, see the NAME option in the description of the RUN[D|V] Command on page 8-156.)
- The backup CPU cannot be the same as the primary CPU.
- The named CPU need not be running at the time that BACKUPCPU is issued.
- If you specify a backup CPU and a backup process already exists, TACL displays an error message.
- If the primary CPU becomes unavailable, TACL switches to the backup CPU. After the primary CPU is reloaded, TACL switches back to the primary CPU. If a user is logged on, TACL postpones this switch till the user logs off.
- You can force your TACL to switch to the backup CPU by using the SWITCH command.
- All events, such as a backup-create error or an I/O error event, and the event details are logged to the primary or $0 collector. This format is used:

  ```
  TACL BACKUP CREATE ERROR: error, DETAIL: error-detail
  ```

  error

  is the error number returned by PROCESS_CREATE.

  error-detail

  is the error detail value returned by PROCESS_CREATE.

  For more information about the $0 collector, see the *EMS Manual*.

- When a backup TACL process takes over, an initial logon state is established. All variables, ASSIGNs, PARAMs, and DEFINEs are reset, and the TACLLOCL file and your TACLCSTM file are invoked. The history buffer index is set to 1.
- If an error occurs while TACL is trying to create the backup process or if the backup CPU is down, TACL waits 3 minutes before trying to create the backup.

Examples

1. Suppose that you are running a TACL process named $CMT1 (the name displayed by the WHO and PPD commands). When you enter the STATUS *, TERM command, you see that $CMT1 does not have a backup process:

   ```
   $E197 10,122 150 001 8,1 $SYSTEM.SYSTEM.TACL $FURD
   ```

   You start a backup TACL process in CPU 11 by entering:

   ```
   12> BACKUPCPU 11
   ```

   TACL displays this response after creating the backup process:

   ```
   BACKUP PROCESS CREATED IN CPU 11
   ```

   Entering “STATUS *, TERM” now displays this information:

   ```
   $E197 10,122 150 001 8,1 $SYSTEM.SYSTEM.TACL $FURD
   $E197 B 11,122 150 R 000 8,1 $SYSTEM.SYSTEM.TACL $FURD
   ```

   This display shows that the primary TACL process (running in processor 10, with process number 122) now has a backup TACL process (running in processor 11, with process number 122). Both the primary and backup processes have a priority of 150. See the STATUS Command on page 8-206 for explanations of the other categories in the STATUS display.

2. To delete this newly created backup process, enter:

   ```
   14> BACKUPCPU
   15>
   ```

   which TACL acknowledges with:

   ```
   STOPPED: 11,122
   BACKUP PROCESS DELETED
   ```

3. To move your backup process from one CPU to another, enter:

   ```
   15>BACKUPCPU
   STOPPED: 11,111
   BACKUP PROCESS DELETED
   16>BACKUPCPU 12
   BACKUP PROCESS CREATED IN CPU 12
   17>
   ```

BREAK Command

Use the BREAK command to set a debugging breakpoint on a specified variable level.

```
BREAK [ variable-level ]
```

variable-level

is the name of an existing variable level or built-in variable.

Considerations

- The specified variable level cannot be in a shared segment.
- If you do not specify a variable level, TACL displays a list of all breakpoints currently set.
- Just before invoking any variable level on which a breakpoint has been set, TACL automatically invokes _DEBUGGER.
- For additional information about how to debug TACL statements, see the DEBUGGER Function on page 8-51.

BUILTINS Command

Use the BUILTINS command to display the names of the TACL built-in functions, built-in variables, or both.

```
BUILTINS [ / {  FUNCTIONS | VARIABLES } / ]
```

FUNCTIONS

displays a list of the built-in functions.

VARIABLES

displays a list of the built-in variables.

Considerations

If you specify neither FUNCTIONS nor VARIABLES, TACL displays all built-in functions and variables.

To obtain the list of built-in functions and variables from within a TACL macro or routine, use the #BUILTINS Built-In Function on page 9-38.

Example

This example illustrates the use of the BUILTINS command:

```
3> BUILTINS / VARIABLES /
```

The built in variables are:

#ASSIGN	#BREAKMODE	#CHARACTERRULES
#DEFAULTS	#DEFINEMODE	#ERRORNUMBERS
#EXIT	#HELPKEY	#HOME
#IN	#INFORMAT	#INLINEECHO
#INLINEOUT	#INLINEPREFIX	#INLINEPROCESS
#INLINETO	#INPUTEOF	#INSPECT
#MYTERM	#OUT	#OUTFORMAT
#PARAM	#PMSEARCHLIST	#PMSG
#PREFIX	#PROCESSFILESECURITY	#PROMPT
#REPLYPREFIX	#SHIFTDEFAULT	#TACLSECURITY
#TRACE	#USELIST	#WAKEUP
#WIDTH		

BUSCMD Program (Super-Group Only)

Use the BUSCMD program to affect the status of the X or Y interprocessor bus (D-series RVUs) or the X or Y ServerNet fabric (S-series RVUs) between two or more processors. You must use a super-group user ID (255,*your-id*) to issue this command.

```
BUSCMD [ / run-option [ , run-option ] ... / ]
  { X | Y } , { DOWN | UP } , from-cpu , to-cpu
```

run-option

is any of the options described in the RUN[D|V] Command on page 8-156.

`{ X | Y }`

specifies the bus or fabric (X or Y) whose status is to be changed.

`{ DOWN | UP }`

specifies whether the bus or fabric is to be made unavailable (DOWN) or available (UP).

from-cpu

is a CPU number in the range from 0 through 15 that is one endpoint of the bus or fabric whose operational status is to be changed. Specify -1 to indicate all processors.

to-cpu

is a CPU number in the range from 0 through 15 that is the other endpoint of the bus or fabric whose operational status is to be changed. Specify -1 to indicate all processors.

Examples

1. A super-group user in a four-processor system can bring down the X bus from processor 1 to all other processors by entering:

```
14> BUSCMD X, DOWN, 1, -1
THE X BUS FROM CPU 01 TO 00 HAS BEEN DOWNED.
THE X BUS FROM CPU 01 TO 01 HAS BEEN DOWNED.
THE X BUS FROM CPU 01 TO 02 HAS BEEN DOWNED.
THE X BUS FROM CPU 01 TO 03 HAS BEEN DOWNED.
```

2. A super-group user can bring the Y bus up from processor 1 to processor 2 by entering:

```
15> BUSCMD Y, UP, 1, 2
THE Y BUS FROM CPU 01 TO 02 HAS BEEN UPPED.
```

CLEAR Command

Use the CLEAR command to delete logical-file assignments made with the ASSIGN command or parameters set with the PARAM command.

```
CLEAR { ALL } | { ALL ASSIGN |
                  ALL PARAM |
                  ASSIGN logical-unit |
                 PARAM param-name }
```

`ALL`

deletes all logical-file assignments and parameters.

`ALL ASSIGN`

deletes all logical-file assignments made with the ASSIGN command.

`ALL PARAM`

deletes all parameters set with the PARAM command.

`ASSIGN` *`logical-unit`*

deletes the assignment for *`logical-unit`*; see the ASSIGN Command on page 8-21 for information about *`logical-unit`*.

`PARAM` *`param-name`*

deletes *`param-name`*; see the PARAM Command on page 8-113 for information about *`param-name`*.

Examples

1. This command deletes all logical-file assignments made with the ASSIGN command and all parameters set with the PARAM command:

   ```
   14> CLEAR ALL
   ```

2. To delete all logical-file assignments made with the ASSIGN command, enter:

   ```
   15> CLEAR ALL ASSIGN
   ```

3. You can delete the assignment for the logical file PRNTFILE and the information associated with it by entering:

   ```
   16> CLEAR ASSIGN prntfile
   ```

4. You can delete the parameter SWITCH-1 and its value by entering:

   ```
   17> CLEAR PARAM switch-1
   ```

CLICVAL Program

Use the CLICVAL program to validate a Core License file.

```
CLICVAL [-help | help ]
```

`-help or help`

displays CLICVAL help text.

CLICVAL validates whether the Core License file is located in the correct location for use by the operating system and is valid for the system. CLICVAL also displays the name of the file that was validated along with the validation results such as OK, Missing, Corrupt, Invalid, Serial Number mismatch, and so on. The CLICVAL program displays the contents of the license file, wherever possible, including the license file version number, the system serial number, and the enabled number of IPUs and CPUs. The contents cannot be displayed if the file is missing or corrupt.

You can use the CLICVAL program on RVUs prior to J06.13, to validate the Core License file prior to upgrading to J06.13 or later from a prior RVU. If you are on J06.13 or a later RVU, you can use the OSM Service Connection action "Read Core License" to validate the Core License file instead. See the *Read Core License action* online help for more information.

For more information about when to use the CLICVAL tool to validate the core license file, see the *NonStop BladeSystem Planning Guide*.

COLUMNIZE Command

Use the COLUMNIZE command to read a list of elements and display those elements in one or more columns, taking into consideration the widest element in the list and the value of the #WIDTH Built-In Variable on page 9-419.

```
COLUMNIZE list
```

list

is a list of elements separated by spaces.

Example

This example has an effect similar to that of the BUILTINS /VARIABLES/ command, displaying a columnar list of TACL built-in variables.

```
10>COLUMNIZE [#SHIFTSTRING [#BUILTINS /VARIABLES/] ]

#ASSIGN          #BREAKMODE          #CHARACTERRULES
#DEFAULTS        #DEFINEMODE         #ERRORNUMBERS
#EXIT            #HELPKEY            #HOME
...
```

TACL adjusts the display to fit the OUT file width. To view the preceding built-in variables in two columns, set the OUT file width to 60:

```
10>#SET #WIDTH 60
11>COLUMNIZE [#SHIFTSTRING [#BUILTINS /VARIABLES/] ]

#ASSIGN          #BREAKMODE          #CHARACTERRULES
#DEFAULTS        #DEFINEMODE         #ERRORNUMBERS
#EXIT            #HELPKEY            #HOME
...
```

COMMENT Command

The COMMENT command causes TACL to ignore the rest of the command line.

```
COMMENT [ comment-text ]
```

comment-text

is any descriptive text you wish to include.

Considerations

- A space is required between COMMENT and *comment-text*.
- In most cases, because they require less processing by TACL, it is preferable to use the comment characters: braces ({ and }) and double equals (==). However, where TACL statements are input from the IN file, such as in OBEY files, you should use the COMMENT command because #INFORMAT must be set to TACL or QUOTED for metacharacters to be recognized as such (but the COMMENT command is always effective).
- COMMENT is a command, interpreted by TACL, although it does not perform an action and returns a null result.

For additional information about comments, see Comments on page 2-10.

_COMPAREV Function

Use the _COMPAREV function to compare one string with another. The _COMPAREV function is an alias for the #COMPAREV built-in function.

```
_COMPAREV string-1 string-2
```

string-1 and *string-2*

are the names of existing variable levels or STRUCT items, text enclosed in quotation marks, or a concatenation of such elements. The concatenation operator is '+' (the apostrophes are required). Variables must not be of type DIRECTORY.

Result

_COMPAREV returns a nonzero value if the contents of the strings are the same, zero if they are different.

Considerations

- You can compare any combination of STRUCTs and STRUCT items with each other. Such comparisons are case-sensitive.
- You can compare any combination of variables that are not of type DIRECTORY or STRUCT and are not STRUCT items. Such comparisons are not case-sensitive.
- You should use _COMPAREV when a variable level contains text with spaces or when you do not want to obtain the contents of a variable level.
- The comparison is not case-sensitive; that is, an uppercase character is equivalent to its lowercase counterpart.
- To compare a text string to a template, use the #MATCH built-in function.

Examples

1. This example illustrates _COMPAREV, using #OUTPUT to display the result:

```
5> PUSH A B
6> #SET A Hello
7> #SET B Goodbye
8> #OUTPUT [_COMPAREV A B]
0
```

2. The next example illustrates the use of _COMPAREV within a routine:

```
#PUSH A B
#SET A Hello
#SET B Goodbye
[#IF [_COMPAREV A B] |THEN|
   #OUTPUT A equals B
  |ELSE|
   #OUTPUT A does not equal B
]
```

COMPUTE Command

Use the COMPUTE command to display the value of an expression.

```
COMPUTE expression
```

expression

is an expression containing integer or string values and one or more operators, as defined in Expressions on page 3-1.

Considerations

- To obtain the value of an expression from within a TACL macro or routine, use the #COMPUTE Built-In Function on page 9-71; it returns the result of the specified arithmetic expression.
- COMPUTE displays the calculation and its arithmetic result. Comparisons and logical operators display -1 if the test is true, 0 if the test is false.
- Because COMPUTE performs integer division, the fraction portion of the result is omitted, as in 3/4 = 0.

Examples

1. This example illustrates simple use of the COMPUTE command:

```
6> COMPUTE 6 + 4
6 + 4 = 10
```

2. Equations are evaluated from left to right as indicated by the precedence for operators. To change the order of evaluation, use parentheses:

```
7> COMPUTE 4 + 5 * 3
4 + 5 * 3 = 19
8> COMPUTE (4 + 5) * 3
(4 + 5) * 3 = 27
```

3. This example shows computations involving relational and logical expressions:

```
9> #PUSH oui,non
10> #SETMANY oui non, 1 0
11> COMPUTE oui < non
oui < non = 0
12> COMPUTE (non OR oui) AND NOT non
(non OR oui) AND NOT non = -1
```

4. This example illustrates string computations:

```
13> #PUSH a b
14> #SET a 1
15> #SET b 01
16> COMPUTE a=b
a=b = -1
17> COMPUTE a '=' b
a '=' b = 0
```

_CONTIME_TO_TEXT Function

Use the _CONTIME_TO_TEXT function to convert a numeric date and time to a textual date and time.

```
_CONTIME_TO_TEXT contime-list
```

contime-list

is the date and time represented as seven numbers in this format:

```
yyyy mm dd hh mm ss hh
```

Result

_CONTIME_TO_TEXT returns the textual date and time representation of the seven-number date and time returned by the #CONTIME Built-In Function on page 9-75.

Example

This example outputs the date and time in a textual format:

```
14> PUSH var
15> SET VARIABLE var [#CONTIME [#TIMESTAMP] ]
16> #OUTPUT [_CONTIME_TO_TEXT [var] ]
November 21, 1992 14:35:20
```

_CONTIME_TO_TEXT_DATE Function

Use the _CONTIME_TO_TEXT_DATE function to convert a numeric date and time to a textual date.

```
_CONTIME_TO_TEXT_DATE contime-list
```

contime-list

is the date and time represented as seven numbers in this format:

```
yyyy mm dd hh mm ss hh
```

Result

_CONTIME_TO_TEXT_DATE returns the textual representation of the date portion of the seven-number date and time representation returned by the #CONTIME Built-In Function on page 9-75.

Example

This example outputs the date in a textual format:

```
14> PUSH var
15> SET VARIABLE var [#CONTIME [#TIMESTAMP] ]
16> #OUTPUT [_CONTIME_TO_TEXT_DATE [var] ]
November 21, 1990
```

_CONTIME_TO_TEXT_TIME Function

Use the _CONTIME_TO_TEXT_TIME function to convert a numeric date and time to a textual time.

```
_CONTIME_TO_TEXT_TIME contime-list
```

contime-list

is the date and time represented as seven numbers in this format:

```
yyyy mm dd hh mm ss hh
```

Result

_CONTIME_TO_TEXT_TIME returns the textual representation of the time portion of the seven-number date and time representation returned by the #CONTIME Built-In Function on page 9-75.

Example

This example outputs the time in a textual format:

```
14> PUSH var
15> SET VARIABLE var [#CONTIME [#TIMESTAMP] ]
16> #OUTPUT [_CONTIME_TO_TEXT_TIME [var] ]
14:35:20
```

COPYDUMP Program

Use the COPYDUMP program to copy a tape dump onto a disk file or to compress an existing disk dump file that is not compressed.

```
COPYDUMP [ / run-option [ , run-option ] ... / ]
   source-file,dest-file
```

run-option

is any of the options described for the RUN[D|V] Command on page 8-156.

source-file

specifies the dump file that is to be copied and compressed. Specify either the name of the tape device where the tape dump file is located or the name of a disk dump file you want to copy.

If *source-file* is a disk file, it must be created by using the RCVDUMP program or the RECEIVEDUMP TACL command. If *source-file* is a tape file, it must be created by performing a tape dump.

dest-file

specifies the destination file of the COPYDUMP operation. Specify the name of a disk file. If this disk file does not exist, it is created during the COPYDUMP operation. If this disk file does exist, it must be empty (0 EOF) and have file code 9614.

Considerations

COPYDUMP is not supported on H-series systems.

You can also use FUP (CREATE and COPY) to copy tape dump files to disk files. But COPYDUMP is faster, and generates a smaller disk dump file because it compresses the dump. Also, COPYDUMP automatically determines the size of the disk dump file, whereas you must specify the extent size of the disk file if you use FUP. The COPYDUMP program usually resides in the file $SYSTEM.SYS*nn*.COPYDUMP.

Examples

If a tape dump file resides on the tape mounted on the tape drive $TAPE2, you can copy and compress the tape dump file into the disk file $DATA.DUMPS.CPU1 with the command:

```
47> COPYDUMP $TAPE2, $DATA.DUMPS.CPU1
```

To compress the disk dump file $BAS10.DUMPS.CPU3 into the disk file $BAS10.CDUMPS.CPU3, enter:

```
48> COPYDUMP $BAS10.DUMPS.CPU3 , $BAS10.CDUMPS.CPU3
```

COPYVAR Command

Use the COPYVAR command to copy the contents of one variable level to another.

```
COPYVAR variable-level-in variable-level-out
```

variable-level-in

is the name of an existing variable level.

variable-level-out

is the name of a variable level. COPYVAR performs an implicit PUSH command to create a variable-level-out with the same data type as variable-level-in, regardless of whether variable-level-out already exists.

Result

The COPYVAR command pushes variable-level-out and copies the contents of variable-level-in to variable-level-out.

Example

This example copies the contents of the variable FIRSTDAY to the variable TEMPDAY.

```
14> OUTVAR firstday
FRIDAY
15> COPYVAR firstday tempday
16> OUTVAR tempday
FRIDAY
```

CREATE Command

Use the CREATE command to create an unstructured disk file. An unstructured disk file can contain a data-record structure that the file system does not recognize, or it can be made up of an array of bytes without data-record structure.

```
CREATE file-name [ , parameter [ , parameter ] ]
```

file-name

is the name of the file to be created. If no volume or subvolume is specified in *file-name*, the current volume name and subvolume name in effect for the TACL process at the time of the request are used.

parameter

can be any of these:

extent-size

is the size of the file extents that can be allocated to the file. Specify extent-size as an integer in the range 1 to 65535 for the number of 2048-byte data pages in each extent. The disk process can allocate up to 16 extents (if the MAXEXTENTS file attribute has not been altered from its default of 16) to any file you create with the CREATE command. The default value for *extent-size* is 2.

PHYSICALVOLUME *physical-volume*

is the physical volume on which the logical *file-name* is to be created. If no physical volume is specified in *file-name,* the default volume name in effect for the TACL process at the time of the request is used.

If a logical volume is specified in *file-name*, the physical volume on which the file is created is chosen by the system. The physical-volume parameter overrides this selection

The PHYSICALVOLUME option should be used only if no physical volume is specified in *file-name*. Otherwise, a file-system error is returned.

Considerations

- The security of the file you create is the current default security at the time you enter the command. See the DEFAULT Program on page 8-53 and VOLUME Command on page 8-256 for information about changing default security.
- You can change the security of a file with the FUP SECURE command. See the *File Utility Program (FUP) Reference Manual* for a description of FUP SECURE.
- FUP also has a CREATE command. When you create a file using the CREATE command in TACL, the size of the first extent allocated (the primary extent) defines

the size of all extents that are subsequently allocated (secondary extents). With the FUP CREATE command, however, you can specify different sizes for primary and secondary extents and can partition files into more than one volume. You can also create files of a type other than unstructured. See the *File Utility Program (FUP) Reference Manual* for a description of the FUP CREATE command.

- If the RVU of the system (system procedure FILE_GETINFOLISTBYNAME_) does not support INCOMPLETESQLDDL, UNRECLAIMEDFREESPACE, PHYSICALVOLUME, or PHYSICALFILENAME, this message is generated:

  ```
  *ERROR* FILE_GETINFOLISTBYNAME_ error = 561
  ```

Examples

1. To create an unstructured disk file named DATAFL in your current default subvolume, enter:

   ```
   14> CREATE datafl
   ```

   Because the default extent size is two pages (4096 bytes), the largest size DATAFL can attain (when the maximum of 16 extents is allocated to the file) is 32 pages.

2. To create an unstructured file named BANANA in the subvolume $DSK1.SVOL, with an extent size of 10,240 bytes (5 pages), enter:

   ```
   15> CREATE $dsk1.svol.banana, 5
   ```

   The largest size BANANA can attain (when the maximum of 16 extents is allocated to the file) is 80 pages.

3. To specify a physical volume name when creating a logical file, enter:

   ```
   >3 CREATE $v.s.f,2,$mg or CREATE $v.s.f,$mg,2
   ```

   Either syntax specifies that logical file “$v.s.f” be created with a two-page extent size on physical volume $mg.

CREATESEG Command

Use the CREATESEG command to create a TACL segment file.

```
CREATESEG file-name
```

file-name

is the name to be assigned to the created segment file.

Considerations

- A segment file is a memory-mapped file that can be loaded rapidly into an extended memory segment. When you log on, TACL creates a default segment file to hold the variables in the root (:) directory. TACL then creates the UTILS directory and attaches the TACLSEGF segment file to it for shared access. The TACLSEGF file contains directories for all products (that are part of a software RVU) that use TACL programs and are available on your system. You can add your own library of variables to the default segment file. If you add variables to the default segment file, you must do so every time TACL re-creates the file.
- Along with the standard TACL segment files, you can create a segment file that contains a library of TACL functions. If you create your own segment file, you can instruct TACL to attach the file to a directory variable and use that directory for reference to the file. Follow these steps:
  1. Use the CREATESEG command to create and initialize a segment file.
  2. Use the ATTACHSEG PRIVATE command to associate the segment file with a directory and to gain write access to the file.
  3. Use the HOME command to change the home directory to your new directory.
  4. Use the LOAD command to load the contents of an existing file of TACL routines into the segment file.
  5. Use the HOME command to change your home directory back to the root (:) directory.
  6. Use the DETACHSEG command to release the segment file from the directory. The DETACHSEG command ensures that all data is copied into the segment file before releasing the file.
  7. Use the ATTACHSEG SHARED command to attach the segment file to a directory so that you can use the functions within the segment file.
  8. Set the #USELIST built-in variable to gain access to the segment file.
- These built-in commands are available:
  - #SEGMENT returns information about the default segment file.

  - #SEGMENTCONVERT converts a segment file from a C00 or C10 format to a format version later than C10 or the reverse.
  - #SEGMENTINFO returns information about a segment in use by your TACL process.
  - #SEGMENTVERSION returns the format (C00/C10 or later) of a segment file.

- You cannot create a segment on a remote system.
- TACL allocates virtual memory on a demand basis for its own private segment. It allocates the first extent of 64 pages (131,072 bytes) at process startup. It then allocates additional virtual memory as needed, in 64 page increments, to a 4 extent maximum of 1024 pages (2,097,152 bytes).
- TACL allocates additional extents as needed; it never deallocates extents.
- You cannot use the segment file until you issue an ATTACHSEG command to gain access to it.
- To display a table of information about all the segment files in use by your TACL process, use the SEGINFO command.
- TACL segment files have file code 440.

Examples

1. This command creates a TACL segment file (file code 440) named MYSEG.

```
8> CREATESEG myseg
9>
```

2. These commands create a segment file called MYSEGFIL, attach the segment file to a directory called :mydir, load a set of routines from a file called $DATA.TACL.ROUTINES, detach the segment from private use (and complete the transfer of data to the segment file), and attach the segment for shared use:

```
8>CREATESEG mysegfil
9>ATTACHSEG PRIVATE mysegfil :mydir
10>HOME :mydir
11>LOAD /KEEP 1/ $DATA.TACL.ROUTINES
12>HOME
13>DETACHSEG :mydir
14>ATTACHSEG SHARED mysegfil :mydir
15>#SET #USELIST [#USELIST] :mydir
```

DEBUG Command

Use the DEBUG command to initiate debugging for a process that is already running. Entering the DEBUG command is one way to invoke the Debug program or the Inspect debugger.

If you do not specify a process ($*process-name* or *cpu,pin*), the Debug and Inspect programs begin debugging the process most recently started by TACL, if that process is still in existence.

For information on the Debug program, see the *Debug Manual*. For information about the Inspect symbolic debugger, see the *Inspect Manual.*

The program DEBUG is not available for use on systems running H-series software. See the Considerations section on page 8-49 for more details.

```
DEBUG [ [\node-name.]{$process-name | cpu,pin } ]
  [ , TERM [\node-name.]$terminal-name ]
```

\node-name

is the system where the process resides.

$process-name

is the name of the process or process pair.

cpu,pin

is the CPU number and process number for the process.

TERM [*\node-name.*]$*terminal-name*

is the new home terminal of the process being debugged. To enter DEBUG or INSPECT commands from a terminal other than the current home terminal of the process being debugged, you must designate that terminal as the new home terminal with the TERM option. You must include the node-name if the new home terminal is connected to a system other than the current default system.

If you omit the TERM option, DEBUG or INSPECT prompts appear on the original home terminal of the process.

Considerations

- You can enter the DEBUG command interactively only, not in an input file or an OBEY file.
- Unless you are a group manager or the super ID, you can debug only those processes with process accessor IDs that match your user ID. You must also have

read access to the program file. (For a description of process accessor IDs, see the *Guardian User's Guide*.)

- A group manager can debug any process whose process accessor ID matches the user ID of any user in the group. The manager must also have read access to the program file.
- The super ID can debug any process. Only the super ID can debug privileged processes.
- The process you name in the DEBUG command does not enter the debug mode until it executes its next instruction in the user code space. The process cannot enter the debug state while executing system code.
- DEBUG is the default debugger. When you enter the DEBUG command, DEBUG is invoked unless INSPECT was designated as the debugger for the process when the process was created. You can designate INSPECT as the default debugger in any of these ways:
  - Issue a SET INSPECT ON command, and then run the program.
  - Use the INSPECT option with the RUN command.
  - When compiling a source program, or when using BINDER, include a compiler directive that specifies that the debugger for the program is to be INSPECT.

H-Series Usage

The program DEBUG is not available for use on systems running H-series software.

The DEBUG command invokes a debugger, it can be Inspect, Native Inspect (eInspect, which is not in the family of Inspect debuggers), or Visual Inspect.

The rules about which debugger gets invoked are approximately the same as for the RUND command. That is, if the INSPECT attribute is set ON anywhere (in the object file during compilation, or on the TACL command line using the SET command), you will get a debugger in the Inspect family (either Inspect or VI), unless of course neither of these debuggers is available, and then you get the default debugger, eInspect. If the Inspect attribute is OFF, you get Native Inspect (eInspect).

Inspect is invoked only for TNS accelerated/interpreted programs (never for TNS/E native programs), while Visual Inspect can handle both of these. Native Inspect handles only TNS/E native programs and snapshots.

Examples

- To debug a running process named $ERRER, enter:

```
14> DEBUG $ERRER
```

  DEBUG or INSPECT then prompts for commands.

- To use the terminal $WHITE to debug the process whose process ID is 8,45, enter:

```
15> DEBUG 8,45, TERM $WHITE
```

  A DEBUG or INSPECT prompt then appears on terminal $WHITE.

DEBUGGER Function

If tracing is enabled (see #TRACE Built-In Variable on page 9-406), the TACL trace facility displays the next line to be executed, then immediately invokes the _DEBUGGER function before executing that code. You can set breakpoints, modify variables, and invoke TACL commands while debugging.

For an example of _DEBUGGER function use, see the *TACL Programming Guide*.

```
_DEBUGGER current-trace-value reason-for-entry
```

current-trace-value

is the value of #TRACE, which always equals -1 (true) when invoked by TACL.

reason-for-entry

is TRACE or BREAK.

Considerations

- When #TRACE is not zero, TACL calls _DEBUGGER immediately after displaying the code that is next to be invoked. In this case, current-trace-value is always -1 and reason-for-entry is always TRACE.
- It is possible to replace _DEBUGGER with a debugging function of your own devising.
- If TACL is about to invoke a variable on which you have set a breakpoint, TACL invokes _DEBUGGER. _DEBUGGER prompts with - num- (the current history number). At this point you can enter a command. If it is a TACL command, it is passed on to TACL for execution. If it is a _DEBUGGER command, _DEBUGGER executes it. The _DEBUGGER commands are:

  B[REAK] [*variable-level*]

  sets an invocation breakpoint on a variable level. If you omit *variable-level*, the command displays all breakpoints currently set.

  C[LEAR] { *variable-level* | * }

  clears the invocation breakpoint for the specified *variable-level*. If you use the asterisk (*), the command clears all breakpoints.

  D[ISPLAY] *variable-level*

  displays the contents of the specified variable level.

  M[ODIFY] *variable-level*

  allows you to modify the contents of the specified variable level.

R[ESUME]

exits the debugger and continues execution of TACL statements.

ST[EP]

invokes the next function, then waits for you to press RETURN before performing the next action. You can continue in this way, stepping through the function.

You can abbreviate each of these commands to the letter or letters outside the square brackets; that is, you can issue DISPLAY as D or STEP as ST.

△ **Caution.** While TACL is in the debugging mode, you can issue any TACL commands that do not conflict with _DEBUGGER commands. Use extreme care in issuing any commands that would affect _DEBUGGER or its environment, including altering a variable level that is being traced. It is unwise to include an #UNFRAME, for instance, because the _DEBUGGER environment would be destroyed

DEFAULT Program

Use the DEFAULT program to set your logon (saved) default system, volume, and subvolume names, and to set your logon disk file security. The logon defaults are in effect whenever you log on.

```
DEFAULT [ / run-option [ , run-option ] ... / ] default-names
  [,"default-security"] ,"default-security"
```

run-option

is any of the options described in the RUN[D|V] Command on page 8-156.

default-names

are the volume and subvolume names that are to be your logon defaults; that is, these are the current defaults when you log on or enter the VOLUME command without any parameters. You can also specify a logon default node name. TACL uses the current default node, volume, and subvolume names to expand partial file names.

The form of default-name is:

```
[\node-name.] volume-or-subvolume-names
```

node-name

is the node name that is to be your default when you log on or enter the SYSTEM command without parameters. If you omit node-name, your logon default system does not change.

volume-or-subvolume-names

is the name of your logon default volume or subvolume or both. These are your current defaults when you log on or enter the VOLUME command without any parameters.

For *volume-or-subvolume-names*, specify one of:

```
$default-volume-name
default-subvolume-name
$default-volume-name.default-subvolume-name
```

$*default-volume-name*

is the name of your new logon default volume. If you omit $*default-volume-name*, the current default volume is your new logon default.

default-subvolume-name

is the name of your logon default subvolume. If you omit *default-subvolume-name*, the current default subvolume becomes your new logon default.

default-security

is the default disk file security that is to be in effect when you log on or enter the VOLUME command without any parameters. (The current default file security is assigned to newly created disk files unless you explicitly assign a different security setting when you create a file.) For *default-security*, specify a string of four characters inside quotation marks; the four characters represent the four security attributes:

R W E P

R

specifies who can read the file.

W

specifies who can write to the file.

E

specifies who can execute the file.

P

specifies who can purge the file.

Each security attribute (R, W, E, and P) can be any of these characters:

O	(Owner) Only the owner can access the file; the owner must be logged onto the local system.
G	(Group) Anyone in the owner's group can access the file; the user must be logged onto the local system.
A	(Anyone) Any user can access the file; the user must be logged onto the local system.
U	(User) Only the owner can access the file; the owner may be logged onto thelocal system or a remote system.
C	(Community) Anyone in the owner's group can access the file; the user may be logged onto the local system or a remote system.
N	(Network) Any user can access the file; the user may be logged onto the local system or a remote system.
-	Only the local super ID can access the file.

Considerations

- During an operating session, you can change the current default values with the SYSTEM and VOLUME commands. TACL uses the current default system, volume, and subvolume names to expand partial file names. When you create a file, it is assigned the default disk file security unless you explicitly specify a

different security. For more information about changing the default values and about disk file security, see the *Guardian User's Guide*.

- Any new logon default values you set with the DEFAULT program do not take effect until the next time you log on or until you enter the VOLUME command with no parameters.
- The current default settings are often different from your saved logon default settings. When you log on, your logon default settings are in effect for system (if you are on a network), volume, subvolume, and file security. After you log on, you can change the current default volume or system with the VOLUME or SYSTEM command. Neither of these commands, however, affects your logon default settings. Every time you log on or enter a VOLUME or SYSTEM command with no parameters, your logon default settings are restored. If you have changed these settings with the DEFAULT program, the new values are in effect. For more information, see the descriptions of the VOLUME Command on page 8-256, and SYSTEM Command on page 8-221.
- The TACL process creates a root segment, even if the default volume doesn't exist. Users can logon even if the default volume doesn't exist.
- Avoid specifying your current local system as the default system: Doing so is unnecessary, and can in fact cause problems with some programs.
- You cannot use the security specifier "-" (allow access to local super ID only) with the DEFAULT program.
- When a new user is added, the logon defaults are:
  - Volume $SYSTEM
  - Subvolume NOSUBVOL
  - Security "AAAA"

Example

Assume that you want your logon default volume to be $BIG, and your logon default subvolume to be BAD. You also want your logon default disk file security to:

- Allow any user on any system to read and execute a file with your default security
- Allow only you, the owner of a file, to write to and purge a file with the default security. (In addition, you must be logged onto the system where the file resides to be able to write to such a file or to purge it.)

To set these defaults, enter:

```
14> DEFAULT $BIG.BAD, "NONO"
THE DEFAULT sys-vol-svol HAS BEEN CHANGED TO $BIG.BAD
THE DEFAULT file-security HAS BEEN CHANGED TO "NONO".
```

To put these new logon defaults into effect, you must either log on again or enter:

```
15> VOLUME
```

DELETE DEFINE Command

Use the DELETE DEFINE command to remove one or more existing DEFINEs from the process file segment (PFS) of the current TACL. For information on DEFINEs, see Section 5, Statements and Programs.

```
DELETE DEFINE define-name-list
```

define-name-list

specifies one or more existing DEFINEs to be deleted. For define-name-list, specify either of these:

```
define-template
( define-template [ , define-template ] ... )
```

define-template

is a DEFINE name that can optionally contain template characters:

* matches zero or more characters

? matches a single character

A DEFINE template that consists entirely of =* or ** causes all existing DEFINEs to be displayed.

Considerations

- The DELETE DEFINE command affects only DEFINEs for the current TACL process. Any DEFINE that was propagated from the current TACL to other processes is unaffected.
- You cannot delete the =_DEFAULTS DEFINE.
- To obtain error information, use the #ERRORNUMBERS Built-In Variable on page 9-160.

Example

To delete the DEFINE named =DFILE, enter:

```
63> DELETE DEFINE =DFILE
```

DELUSER Program (Group Managers Only)

Use the DELUSER program to delete users from a system. To use the DELUSER program, you must have a user ID of *n*,255.

```
DELUSER [ / run-option [ , run-option ] ... / ]
  group-name.user-name
```

run-option

is any of the options described in the RUN[D|V] Command on page 8-156.

group-name.user-name

are the group and user names of the user who is to be deleted. Each name can contain from one to eight letters or digits. The first character must be a letter.

Consideration

Group managers can delete only those users who are members of their respective groups. The super ID can delete any user in the system.

Example

The group manager of the BIG group (or the super ID) can delete the user BIG.BOZO (user ID 7,12) by entering:

```
14> DELUSER BIG.BOZO
BIG.BOZO (7,12) HAS BEEN DELETED FROM THE USERID FILE.
```

DETACHSEG Command

Use the DETACHSEG command to relinquish use of a segment file.

```
DETACHSEG directory-name
```

directory-name

is the name of a directory whose variables are in another segment file.

Considerations

- Executing the DETACHSEG command is equivalent to popping the directory.
- The directory must be the top level of its variable because detaching a segment pops the segment.
- The DETACHSEG command does not allow you to relinquish use of the segment file whose root is :UTILS because you would lose all your commands if you did so.
- You cannot detach the default private segment file created automatically by TACL when you log on. This is the segment file that contains the root (:).
- Current or pushed HOME directories that are in the segment being detached are set to the root directory.
- Current or pushed use-list directories that are in the segment being detached are removed from those use lists.
- Any #REQUESTER having any of its variables in the segment being detached is closed.
- Any server, implicit or explicit, having any of its variables in the segment being detached is deleted; its openers receive error 66 on all subsequent I/O to the server.
- Active macros and routines whose code is stored in variables in the segment being detached run to completion before detachment occurs.
- When you detach a private segment file:
  - All segment files attached to it are detached from it. If you later attach the segment file again, those segment files are not automatically attached again.
  - All variable levels within the segment are set to frame 0. This does not destroy any variable levels, but gives the appearance that all variable levels were created in frame 0.
  - There is a delay while TACL writes the segment file. TACL writes only as much as has been used, not the whole file.
  - All breakpoints in that segment file are cleared.

- If the CPU on which your TACL process is running fails while you are in the act of detaching a private segment file, the segment file may be corrupted. Any future attempts to attach that file will fail. You must purge it and re-create it.

Note. ATTACHSEG operates by pushing and defining a directory variable that refers to the specified segment file; DETACHSEG operates by popping that directory variable. Because #UNFRAME pops all variables pushed since the most recent #FRAME, if you attach a segment file following a #FRAME, the corresponding #UNFRAME detaches the segment file; its contents are no longer available. Subsequent attempts to invoke those contents result in errors.

Example

This command detaches the segment file that is attached to the directory variable MYDIR:

```
25> DETACHSEG :mydir
```

ENV Command

Use the ENV command to display the settings of one or all the environmental parameters for your TACL.

```
ENV [ environment-parameter ]
```

environment-parameter

is one of these:

```
HOME
INLECHO
INLOUT
INLPREFIX
INLTO
PMSEARCH
SYSTEM
USE
VOLUME
```

HOME

displays the name of your current home directory. You set this value with the HOME command.

INLECHO

displays the setting, OFF or ON, of the #INLINEECHO Built-In Variable on page 9-200, which controls whether TACL copies to its own OUT file the lines it sends as input to inline processes.

INLOUT

displays the setting, OFF or ON, of the #INLINEOUT Built-In Variable on page 9-202, which controls whether TACL copies to its own OUT file lines sent by inline processes to their OUT files.

INLPREFIX

displays the (possibly null) value of the #INLINEPREFIX Built-In Variable on page 9-203, which contains the character or characters constituting the prefix that identifies lines to be passed to inline processes.

INLTO

displays the (possibly null) value of the #INLINETO Built-In Variable on page 9-206, which contains the name of a variable to which are appended copies of lines sent by inline processes to their OUT files, if such a variable has been defined.

PMSEARCH

displays the search list that your TACL uses to find program and macro files when they are invoked implicitly. This list is set with the PMSEARCH Command on page 8-118.

SYSTEM

displays the (possibly null) name of your current default system. You set and clear this value with the SYSTEM Command on page 8-221.

USE

displays the directories that TACL searches for variables not found in your home directory. You set this list with the USE Command on page 8-231.

VOLUME

displays the name of your current default volume, including the name of your current default system if it is different from your saved default system. Your current default file security is also defined. You set these values with the VTREE Command on page 8-258.

Consideration

If you do not specify any parameter, ENV displays the settings of all your TACL environment parameters.

Examples

- To display your current default system, volume, subvolume, and file security, enter:

```
14> ENV VOLUME
Volume \MYSYS.$WORK.PROJECT, "NUNU"
```

- To display all the environment parameters for your TACL process, enter ENV. This output is returned:

```
15> ENV

Home              :MYDIR
Inlecho           OFF
Inlout            ON
Inlprefix         #
Inlto
Pmsearch          #DEFAULTS, $SYSTEM.SYSTEM
System
Use               :MYDIR.1, :, :UTILS.1, :UTILS.1:TACL.1
Volume            \MYSYS.$WORK.PROJECT, "NUNU"
```

EXIT Command

Use the EXIT command to stop the current TACL process or TACL process pair. Pressing CTRL-y is the same as typing EXIT.

```
EXIT
```

Considerations

- To use the EXIT command to delete a TACL process on your home system, you must be logged onto that system. You can, however, use EXIT to delete a TACL process you started on a remote system without logging onto the remote system.
- If TACL encounters an EXIT command or an end-of-file (EOF) while executing an OBEY command file, command execution halts and control returns to the TACL process in which the OBEY command was entered.
- If TACL encounters an EXIT command while executing commands from an input file (such as an IN file named in the command to start a TACL process), and TACL is not running interactively, the TACL process is deleted. Control returns to the process from which TACL was started.
- If you enter an EXIT command from the terminal when both IN and OUT refer to that terminal (the interactive mode), TACL asks:

```
Are you sure you want to stop this TACL ( process-spec)?
```

To stop the current TACL, enter Y, YE, or YES followed by a RETURN. This may make your terminal unavailable to the system (see the note, below). If you enter any other character or only a RETURN, TACL ignores the EXIT command.

Note. If you delete your last TACL process, you cannot communicate with the system from your terminal. You can, however, use another terminal to start a new TACL process on your terminal. (See the syntax description for the TACL Program on page 8-224.)

FC Command

Use the FC command to retrieve, edit, and reexecute lines in the history buffer.

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

num

is a positive integer that refers to the specific command-line number you want to retrieve from the history buffer.

- num

is a negative number that refers to a command line in the history buffer relative to the current command line.

text

is the most recent command line in the history buffer that begins with the text you specify. This text need only be as many characters as necessary to identify the command line.

Consideration

When you enter the FC command, the specified command line is redisplayed (if you do not specify a line, the most recent command line is redisplayed). A double period prompt (..) appears below it.

Examples

Suppose that you are renaming the file JUNK to TESTFILE, and make a typing error in the RENAME command on line 9. Rather than retype the whole line, type FC 9 to fix the invalid command:

```
12> FC 9
12> RENMAE JUNK, TESTFILE
12..
```

You now see your invalid command redisplayed on the screen. The blank line at the double period prompt is an editing template. In this template, you enter subcommands that make corrections or additions to the command displayed above the template. The rules for entering subcommands and for making corrections and additions are described for the Editing Template on page 8-64.

After you enter the subcommands you want, press the RETURN key. Your edited command and a new editing template then appear. If you want to make more corrections to the edited command, you can enter more subcommands in the editing template and then press RETURN. If the edited command is correct, you can reexecute it by pressing RETURN without entering any subcommands.

To correct the spelling error in this example, you could enter:

```
12> RENMAE JUNK, TESTFILE
12.. AM
12> RENAME JUNK, TESTFILE
12..
```

Editing Template

The syntax of the editing template is:

```
subcommand [ // subcommand ] ...
```

subcommand is any of these:

`{ R | r }` *replacement-text*

`{ I | i }` *insertion-text*

`{ D | d }`

replacement-text

//

is a separator, allowing multiple subcommands on a given line. A subcommand can immediately follow one or more uppercase or lowercase D's without being preceded by //.

`{ R | r }` *replacement-text*

replaces characters in the previous command, starting with the character displayed immediately above the R or r. A replacement-text preceded by R or r can be any string of characters, including spaces, and can itself begin with R, I, or D (or r, i, or d). Characters in replacement-text replace characters in the previous command on a one-for-one basis.

If // follows this subcommand, all characters in replacement-text up to //, including spaces, replace characters in the previous command. Otherwise, replacement ends with the RETURN.

`{ I | i }` *insertion-text*

inserts characters into the previous command in front of the character displayed above the I or i. If // follows this subcommand, all characters in insertion-text up to //, including spaces, are inserted into the previous command. If no // appears, all characters up to the RETURN are inserted.

`{ D | d }`

deletes characters in the previous command. Any original character displayed above a D or d that begins a subcommand in the editing template is deleted.

replacement-text

is any text that does not begin with R, I, or D (or r, i, or d). Characters in replacement-text replace characters immediately above them on a one-for-one basis. For example, a D in replacement-text replaces the character displayed above it instead of deleting it.

Considerations

- If you enter FC alone, the last command entered is displayed.
- You can use the FC command before you are logged on. For example, FC works to correct errors in the LOGON command. If your password contains control characters, however, FC might not edit them correctly.
- If you press the BREAK key before completing an editing template, the FC command is aborted, and the previous command is not reexecuted. The original command is left unchanged.
- If you enter only the subcommand separator (//) in the editing template and follow it immediately with a RETURN, the FC command aborts, leaving the original command unchanged.
- Text arguments are not case-sensitive.
- If you try to retrieve a line that is not in your history buffer, you receive an error.
- FC must be entered from the IN file (normally your terminal); it cannot be included in a macro, for example. Similarly, you cannot change FC to another name with an ALIAS, nor can you program a function key to execute FC.

Examples

1. This example demonstrates the use of subcommands D, R, and I and the subcommand separator (//):

```
11> COMMENT This are a commnt
12> FC
12> COMMENT This are a commnt
12.. DRis// Ie
12> COMMENT This is a comment
```

2. This example retrieves the command line numbered 5 in your history buffer:

```
13> FC 5
13> WHOM
13..
```

3. This example retrieves the third command back from your current command-line number:

```
14> FC -3
14> COMMENT This are a commnt
14..
```

4. This example retrieves the most recent line in your history buffer that begins with the text “STA”:

```
15> FC STA
15> STATUS *, TERM
15..
```

FILEINFO Command

Use the FILEINFO command to display information about one or more disk files. The FILEINFO command allows the use of a file-name template. The FILEINFO command is an alias for the #XFILEINFO Built-In Function on page 9-420.

```
FILEINFO [ / OUT list-file / ]
  [ file-name-template [ [,] file-name-template ] ... ]
```

`OUT list-file`

specifies a device, or a sequential file accessible to the sequential I/O (SIO) facility, that is to receive the output from FILEINFO. If you omit this option, TACL writes the listing to its current OUT file.

If you specify an OUT file that does not exist, TACL creates an EDIT file named list-file. If you specify an OUT file that already exists, TACL appends the information to the end of the file.

file-name-template

is a list of one or more disk file specifications separated by commas or spaces. Each disk file specification must be formatted as follows:

`[[[\node.]$volume.] subvolume.] file-name ]`

If, however, you specify a volume name, you must specify a subvolume name. This file name is not syntactically correct:

`$volume-name.file-name`

For a description of filename templates, see File-Name Templates on page 2-7.

Considerations

If you do not specify a file-name template, FILEINFO displays information about all files in the default volume and subvolume.

When a file is copied from one system to another system in a different time zone without modifying the source date of the file, the FILEINFO command returns the file modification time as the local civil time of the system from which the file was copied.

FILEINFO displays information in a form similar to the following:

```
$SPOOL.LINDA

               CODE              EOF   LAST MODIFIED  OWNER  RWEP   PExt   SExt
AFILE             0                0 07DEC2000 15:58   8,185 NUNU      2      2
```

showing the name of each file requested, one or more letters identifying the file status, its file code, the size of the file, the date and time when it was last modified, the user ID

of its owner, its file security, and the number of primary and secondary extents allocated to it.

If the request is for a file that does not exist, the TACL process returns an error message:

```
No files match \TSII.$SPOOL.LINDA.BFILE
```

If the request is formatted as $*volume.filename*, the TACL process returns an error message:

```
FILEINFO $SPOOL.A
         ^
Expecting /
Or a valid filename template
Or end
```

The command output display provides this information:

Character Position in Output Display	Field Description	Field Value
[1 - 8]	filename	Name of the file whose characteristics are being displayed, left justified
[9 - 10]	blank	
[11]	CODE: *Open File or Crash-Open File Indicator*	*" "* File is not open, crashed, or broken. "O" File is open. "?" File was open during a crash and therefore the content might have been compromised.
[12]	CODE: *Corrupt File or SQL DDL Free Space Status Indicator*	"C" File is corrupt; that is, the file content has been compromised. Some type of media recovery operation is required before the file can be accessed. "D" An SQL DDL operation is currently in progress. "F" An SQL DDL operation has left unreclaimed free space in the file.
[14 - 18]	CODE: *File Code*	" " Default file code is assigned to user-created files. "OSS" An OSS file. "100 through 999" Reserved - See system file code definitions, in the *File Utility Program (FUP) Reference Manual* to decipher the code. These codes include the new native object file for TNS/E, the 800 file code. All other values would be explicitly assigned as part of file creation by the user.

Character Position in Output Display	Field Description	Field Value
[19 - 22]	CODE: *FLAGS*	"A" File is audited by TMF. "L" File is licensed. See file licensing information in the *File Utility Program (FUP) Reference Manual*. “P” The PROGID attribute of the file is on. See file PROGID information in the *File Utility Program (FUP) Reference Manual*. "+" File is a format 2 file. See format 2 file information in the *File Utility Program (FUP) Reference Manual*.
[23 - 35]	EOF	“0” File is empty. “*n* through *nnnnnnnnnnnn*” File size in bytes "**********" File size display exceeds 13 digits.
[36]	blank	
[37 - 45]	LAST MODIFIED: *day, month, year*	“*ddMMMyyyy”* The numeric day of the month, three-letter abbreviation of the month, and calendar year that the file was last written.
[46]	blank	
[47 - 51]	LAST MODIFIED: *hour, minute*	The hour and minute when the file was last written
[52]	blank	
[53 - 59]	OWNER *ggg,uuu*	The group number and user number that identify the file owner Note: The super ID (255,255) is displayed as -1.

Character Position in Output Display	Field Description	Field Value
[60]	blank	
[61 - 64]	RWEP	File security assigned to the file for read, write, execute, and purge access. "****" File access is controlled with Safeguard security. "####" File access is controlled with OSS security. “*xxxx*” File access is controlled with Guardian security: “-” local super ID only “O” local owner only “G” local member of owner's group only “A” any local user “U” local or remote owner “C” local or remote member of owner's group “N” any local or remote user For SQL/MX objects, the security options RWEP are displayed as *SQL. See file security information in the *File Utility Program (FUP) Reference Manual*.
[65]	blank	

Examples

1. This example illustrates the use of FILEINFO with a fully qualified file name:

```
15> FILEINFO \SOFTW.$BOOKS.TACL.V1SEC2B
```

2. The next example illustrates the use of the file-name template characters. The asterisk (*) in the subvolume indicates that you want TACL to search for any subvolume that begins with BOOK. The question mark (?) in the file name indicates that you want information about any file whose name has SEC in the first three positions, any character in the fourth position, and a 2 in the fifth position.

```
16> FILEINFO BOOK*.SEC?2
```

3. An even more generalized command is:

```
17> FILEINFO $VOL*A.*.SEC*2
```

   which searches for any volume whose name begins with $VOL and ends with A, all subvolumes on those volumes, and any file in those subvolumes whose name begins with SEC and ends with 2. Note that in the previous example, SEC02, SEC12, SECT2, and so on, meet the search criteria; in this example, SEC2, SECT02, and SECT9992 also qualify.

FILENAMES Command

Use the FILENAMES command to display the names of all files that satisfy the file-name specifications in the *file-name-template*. The FILENAMES command is an alias for the #XFILENAMES built-in function.

```
FILENAMES [ / OUT list-file / ]
  [ file-name-template [ [,] file-name-template ] ... ]
```

OUT *list-file*

specifies a device, or a sequential file accessible to the sequential I/O (SIO) facility, that is to receive the output from FILENAMES. If you omit this option, TACL writes the listing to its current OUT file.

If you specify an OUT file that does not exist, TACL creates an EDIT file named list-file. If you specify an OUT file that already exists, TACL appends the information to the end of the file.

file-name-template

is a list of one or more disk file specifications separated by commas or spaces. Each disk file specification must be formatted as follows:

```
[[[\ node-name.]$ volume-name.] subvolume-name.] file-name
```

For a description of file-name templates, see Section 2, Lexical Elements.

Consideration

If you do not include any file-name template, FILENAMES lists all files in your default volume and subvolume.

Examples

This example illustrates the output of the FILENAMES command:

```
17> FILENAMES $BOOKS.TACL.SEC*
$BOOKS.TACL
SEC01 SEC02A SEC02B SEC02C SEC02D
18>
```

The asterisk in the file-name template instructs FILENAMES to display the names of all files that begin with SEC, regardless of which, or how many, characters follow.

This example illustrates the use of two file-name template characters. The asterisk (*) in the subvolume indicates that you want TACL to search for any subvolume that begins with BOOK. The question mark (?) in the file name indicates that you want to

list the name of any file whose name has SECT in the first four positions, any character in the fifth position, and a 2 in the sixth position:

```
18> FILENAMES $smith.book*.sect?2
$SMITH.BOOK1
SECT02
$SMITH.BOOK2
SECT02 SECT12
19>
```

FILES Command

Use the FILES command to display the names of files in one or more subvolumes.

```
FILES [ / OUT list-file / ]
  [ subvol-template [ [,] subvol-template ] ... ]
```

`OUT list-file`

specifies a device, or a sequential file accessible to the sequential I/O (SIO) facility, that is to receive the output from FILES. If you omit this option, TACL writes the listing to its current OUT file.

If you specify an OUT file that does not exist, TACL creates an EDIT file named *list-file*. If you specify an OUT file that already exists, TACL appends the information to the end of the file.

subvol-template

is a list of one or more disk subvolume specifications separated by commas or spaces. Each subvolume specification must be formatted as follows:

```
[[[\node-name.]$volume-name.] subvolume-name ]
```

For a description of subvolume-name templates, see Subvolume Templates on page 2-7.

Consideration

If you do not include any subvol-template, FILES lists all files in your default volume and subvolume.

Examples

1. You can list the files in your current default subvolume by entering:

   ```
   34> FILES
   ```

2. To list the files in all subvolumes in your current default volume, enter:

   ```
   35> FILES *
   ```

3. To list the files in all subvolumes that begin with SYS in the volume $SYSTEM, enter:

   ```
   36> FILES $SYSTEM.SYS*
   ```

4. To list the files in subvolume SUBZ in volume $DISC1 of the system \ROME, enter:

   ```
   37> FILES \ROME.$DISC1.SUBZ
   ```

FILETOVAR Command

The FILETOVAR command copies data from a file and appends it to a variable level.

```
FILETOVAR file-name variable-level
```

file-name

is the name of an existing file from which data is to be copied. It must be readable by the sequential I/O facility (SIO). Both format1 and format 2 files are supported.

variable-level

is the name of an existing variable level to which the copied data is to be appended. The variable must not be in a shared segment and must not be a directory, a STRUCT, or a STRUCT item.

Considerations

Lines longer than 239 characters in the file are truncated to 239 characters when they are appended to the variable level.

File I/O appears in PLAIN format, which means that no special interpretation is given to the TACL metacharacters [,], |, ==, {, and } when they are read. In particular, this means that you cannot copy TACL statements from a file to a variable unless you wrote the statements into the file with the VARTOFILE command originally. (When TACL loads code from a file into a variable, it processes metacharacters to produce executable code, but FILETOVAR only moves text into the variable.)

Another, faster way to append data to a variable level is to use the #SET command:

```
#SET /IN file-name/ variable
```

As with the FILETOVAR command, TACL INFORMAT must be set to PLAIN when using the #SET function in this manner.

HELP Command

The HELP command displays a single screen of general information about TACL.

```
HELP
```

HISTORY Command

Use the HISTORY command to display the most recently executed command lines.

```
HISTORY [ num ]
```

num

is the number of commands to display. By default, *num* is 10.

Considerations

- The HISTORY command displays the specified number of lines in the history buffer. The history buffer is 1000 characters long and contains zero or more lines. Each line stored in the history buffer requires as many bytes as the line contains, plus one extra byte.
- The lines TACL retains in its history buffer are those nonblank lines you type in, not the lines TACL executes.
- Each line that an FC command or exclamation point command causes to be repeated becomes a new line in the history buffer, just as if you had typed the command again.
- If you type a line too long to fit into the history buffer, TACL executes the line properly but omits it from the buffer.
- If the number is larger than the history buffer, TACL returns the number of available lines.
- If the number is too large, TACL returns an error.

Example

This example illustrates how to list the last five command lines. TACL lists the previous five lines, including the HISTORY command:

```
32> HISTORY 5
28> ENV
29> FILEINFO BOOK*.SEC
30> SET DEFINE CLASS MAP
31> SET DEFINE FILE RSLTS
32> HISTORY 5

33>
```

HOME Command

Use the HOME command to specify the first directory in which your TACL process searches for variables before searching any other directories.

```
HOME [ directory-name ]
```

directory-name

is the name of an existing variable level of type DIRECTORY. If omitted, the root (:) is assumed.

Considerations

- TACL is said to be “creating a variable” when using (#)PUSH, #DEF, or (#)LOAD. Unless the name of a variable to be created begins with a colon, TACL tries to create the variable in the home directory.
- If this directory is in a shared segment file, TACL cannot create any variables in it.
- You can use the ENV Command on page 8-60 or the #HOME Built-In Variable on page 9-191 to display the value set by this command.
- You can save and restore the setting of HOME by pushing and popping #HOME.
- If you detach a segment containing the current or pushed HOME directory, that HOME directory is set to the root.
- For additional information about directories and segment files, see the CREATESEG Command on page 8-46.

Example

This command establishes the directory variable ANOTHER_DIR, within the directory variable MYDIR, as the home directory:

```
24> HOME :mydir:mykeys
```

To list the home directory, you can use the ENV command. It returns this output:

```
25> ENV

Home              :MYDIR:mykeys
Inlecho           OFF
Inlout            ON
Inlprefix
Inlto
Pmsearch          $SYSTEM.SYSTEM, $DATA.MYFILES, #DEFAULTS
System
Use               :, :UTILS.1, :UTILS.1:TACL.1
Volume            $DATA.MYFILES, "AAAA"
```

INFO DEFINE Command

Use the INFO DEFINE command to display the attributes and values associated with one or more existing DEFINEs in the process file segment (PFS) of the current TACL process. For more information about DEFINEs and their usage, see the *Guardian User's Guide*.

```
INFO [ / OUT list-file / ] DEFINE define-name-list
  [ , DETAIL ]
```

`OUT` *`list-file`*

specifies a device, or a sequential file accessible to the sequential I/O (SIO) facility, that is to receive the listing from INFO DEFINE. If you omit this option, TACL writes the listing to its current OUT file.

If you specify an OUT file that does not exist, TACL creates an EDIT file named *`list-file`*. If you specify an OUT file that already exists, TACL appends the information to the end of the file.

`define-name-list`

is a list of one or more existing DEFINEs whose attributes and values you want to display. For define-name-list, specify either of these:

`define-template`
`(` *`define-template`* `[ , ` *`define-template`* `] ... )`

`define-template`

is a DEFINE name that can optionally contain template characters:

* matches zero or more characters

? matches a single character

A DEFINE template that consists entirely of =* or ** causes all existing DEFINEs to be displayed.

`DETAIL`

specifies that the listing is to include the entire contents of each DEFINE specified by *`define-name-list`*. That is, all the attributes as well as the values themselves are displayed.

Considerations

- If you omit the DETAIL option, the INFO DEFINE command displays only three items: the name of the DEFINE, its CLASS, and an attribute that depends on the class. If you include the DETAIL option, the display lists every attribute that has a value, along with that value. The examples that follow show both types of display.

- You cannot display the names of attributes to which no values have been assigned.
- You can display all DEFINEs by entering INFO DEFINE ** or INFO DEFINE =*. DEFINEs are displayed in ASCII sort sequence by name.
- To obtain error information, use the #ERRORNUMBERS Built-In Variable on page 9-160.

Examples

1. To display the CLASS and the principal attribute of all DEFINEs enter:

```
13> INFO DEFINE **
    Define Name          =ACK
    CLASS                MAP
    FILE                $BILL.THECAT.ACKPHTH

    Define Name          =_DEFAULTS
    CLASS                DEFAULTS
    VOLUME              $BILL.THECAT
```

2. This command displays the principal attributes of all the DEFINEs with names that match a template pattern:

```
14> INFO DEFINE =A*
    Define Name          =ACK
    CLASS                MAP
    FILE                $BILL.THECAT.ACKPHTH
```

3. To display all the attributes (that have values) of an existing DEFINE, enter:

```
15> INFO DEFINE =TEST3, DETAIL
    DEFINE NAME         =TEST3
    CLASS               TAPE
    VOLUME              SCRATCH
    LABELS              OMITTED
    USE                 IN
    DEVICE              $TAPE2
    MOUNTMSG           "Transferring files, low pri.-Fred"
```

INITTERM Command

Use the INITTERM (initialize terminal) command to reinstate the default attributes of your home terminal. The INITTERM command is an alias for the #INITTERM Built-In Function on page 9-199.

```
INITTERM
```

Considerations

Typically, you use the INITTERM command when an application program has left your terminal in an abnormal state. INITTERM calls the SETMODE procedure, function 28.

For information about setting device attributes, see the description of SETMODE settings for terminals in the *Guardian User's Guide* and the description of the SETMODE procedure in the *Guardian Procedure Calls Reference Manual*.

INLECHO Command

Use the INLECHO command to specify whether lines sent as input to an inline process are to be copied to the current TACL OUT file. (Use of the INLINE facility is described in the *TACL Programming Guide*.)

```
INLECHO { OFF | ON }
```

`OFF`

disables echoing of input lines.

`ON`

enables echoing of input lines to the TACL OUT file.

Considerations

- The INLECHO command offers a simplified method of setting the #INLINEECHO Built-In Variable on page 9-200.
- You can display the current setting for INLECHO with the ENV command.
- You can save the previous echo-control setting by pushing #INLINEECHO; you can revert to that setting by popping #INLINEECHO.
- Lines copied to the TACL OUT file are preceded by the prompt that the inline process issued when it requested the line.

INLEOF Command

Use the INLEOF command to send an end-of-file indication to an inline process. (Use of the INLINE facility is described in the *TACL Programming Guide*.)

```
INLEOF
```

Considerations

- The INLEOF command is a simplified interface to the #INLINEEOF Built-In Function on page 9-201.
- TACL waits until the inline process accepts the end-of-file and either prompts again or terminates.

INLOUT Command

Use the INLOUT command to specify whether lines written by inline processes to their OUT files are to be copied to the current TACL OUT file as well. (Use of the INLINE facility is described in the *TACL Programming Guide*.)

```
INLOUT { OFF | ON }
```

`OFF`

disables copying of output lines.

`ON`

enables copying of output lines to the TACL OUT file.

Considerations

- To permit OUT file copying, the inline process must have been started with neither the OUT option nor the OUTV option.
- The INLOUT command offers a simplified interface to the #INLINEPREFIX Built-In Variable on page 9-203.
- You can display the current setting for INLOUT with the ENV command.
- You can save the previous copy-control setting by pushing #INLINEOUT; you can revert to that setting by popping #INLINEOUT.
- CONTROLs and SETMODEs issued by an inline process are ignored.

INLPREFIX Command

Use the INLPREFIX command to establish the prefix that identifies lines that are to be passed as input to inline processes. (Use of the INLINE facility is described in the *TACL Programming Guide*.)

```
INLPREFIX [ prefix ]
```

prefix

is one or more characters constituting a prefix that, when placed at the beginning of a TACL input line and followed by a space, identifies the remainder of the line as input to be passed to an inline procedure instead of being acted upon by TACL.

Considerations

- The INLPREFIX command offers a simplified method of setting the #INLINEPREFIX Built-In Variable on page 9-203.
- If you omit prefix, INLPREFIX sets the inline prefix to its default, a null value (no prefix). For efficiency, it is a good idea to leave the prefix set to null when you are not using the INLINE facility.
- You can display the current setting for INLPREFIX with the ENV command.
- You can save the previous prefix value by pushing #INLINEPREFIX; you can revert to that prefix by popping #INLINEPREFIX.
- You must not use #SET as a prefix because doing so would prevent you from changing the prefix to anything else. Other problems could arise from using other TACL entities as prefixes.
- The prefix must not include a space or an end-of-line.
- The prefix is not case-sensitive.
- When used to identify an input line, the prefix must be followed by a space unless it appears alone on a line. In that case, TACL passes a blank line to the inline process.
- If a prefixed line appears when no inline process exists, an error occurs.

Example

This command establishes “@@” as the inline prefix:

```
327> INLPREFIX @@
```

INLTO Command

Use the INLTO command to establish a variable level that is to receive copies of lines written by inline processes to their OUT files. TACL appends copied lines to the end of the variable level. (Use of the INLINE facility is described in the *TACL Programming Guide*.

```
INLTO [ variable-level ]
```

variable-level

is the name of an existing variable level.

Considerations

- To permit OUT file copying, the inline process must have been started with neither the OUT option nor the OUTV option.
- The INLTO command offers a simplified method of setting the #INLINETO Built-In Variable on page 9-206.
- You can display the current setting for INLTO with the ENV command.
- You can save the previous output variable identity by pushing #INLINETO; you can revert to that output variable by popping #INLINETO.
- *variable-level* must not be a DIRECTORY, a STRUCT, or a STRUCT item.
- If you omit *variable-level*, the copying feature is disabled.

Example

This command defines KEEPER as the variable level to receive inline process output.

```
329> INLTO keeper
```

IPUCOM Program

Use the IPUCOM program for a command line interface which can be used to set, reset, or display an IPU number associated with a process. It can also be used to set or display CPU-wide controls.

Note.

- The keywords in each of the command line syntaxes such as -pname, -bname, -pin and so on are case-insensitive.
- The numeric values (pin, cpu number, ipu number) must be in decimal format.
- The maximum command line length supported by IPUCOM is 529 characters.

Syntax to set an IPU association:

```
IPUCOM {-pname <name> | -bname <name> | -pin <pin> } [-cpu
<cpu-number>] <ipu-number>
```

Either a process name or a CPU pin can be used to associate a process with an IPU. If the *-cpu* option is omitted, then the CPU where the IPUCOM program is running is used. To ensure that the intended process is set, it is recommended to use the *-cpu* option with *-pin*. This also helps a $CMON process to change the CPU assignment when IPUCOM is running.

`-pin <pin>`

specifies the PIN of the process whose association is to be set.

`-pname <name>`

specifies the primary process named <*name*> for a process pair.

`-bname <name>`

specifies the backup process named <*name*> for a process pair.

Remember that the primary and backup processes are on different CPUs. To set an association betweem them requires two invocations of IPUCOM. For processes that are not a part of a process pair, use either the *-pname* or *-bname* option to specify the process name. When *-pname* or *-bname* is used with the *-cpu* option, the process specified is the process named <*name*> in the specified CPU regardless of its primary or backup status.

The specified <*name*> or <*pin*> and <*cpu-number*> are all relative to the system where IPUCOM is running.

Specifying a system name in <*name*> (for example: \MYSYS.$XYZ) is not allowed and results in an error from IPUCOM. IPUCOM expects only a local process name in <*name*>.

Syntax to reset an association:

```
IPUCOM {-pname <pname> | -bname <bname> | -pin <pin>} [-cpu
<cpu-number>] unbind
```

Syntax to display an association:

```
IPUCOM {-pname <pname>|-bname <bname>| -pin <pin> } [-cpu
<cpu-number>] info
```

The IPU associated with the process is displayed, which may dynamically change if the IPU assignment is not currently set via IPUCOM or IPUAFFINITY_SET_.

Syntax to set CPU wide controls:

```
IPUCOM [-balanceSA {on|off}] [-balanceDP2 {on|off}] [-
initialState] [-cpu <cpu-number>]
```

`-balanceSA option`

indicates the process scheduler to enable (**on**, default) or disable (**off**) the balancing of Soft Affinity processes.

`-balanceDP2 option`

indicates the process scheduler to enable (on, default) or disable (off) the balancing of DP2 processes.

`-initialState option`

indicates the process scheduler to put the balancing back to the initial state of the scheduler, and to clear all associations established via IPUAFFINITY_SET_.

`<cpu-number>`

indicates the CPU on which to apply the change. A value of -1 indicates that the change should be applied to all CPUs.

Syntax to display the current IPU control settings:

```
IPUCOM [-cpu <cpu-number>] info
```

The IPU control settings for the specified CPU are only displayed. If a CPU value of *-1* is specified, the IPU control settings for all the CPUs are displayed. If the *-cpu* option is omitted then the CPU where the IPUCOM program is running is displayed.

Syntax to display help text:

```
IPUCOM [help]
```

Considerations

- The IPUCOM program can be run without super-group or other privileges, but to alter any of the process or CPU-wide settings, the user must conform to the security requirements of a corresponding call to IPUAFFINITY_SET_ or IPUAFFINITY_CONTROL_.
  See the *Guardian Procedure Calls Reference Manual* for details on the requirements of those interfaces.
  Note that in particular the user of IPUCOM must be locally logged in to alter any of the process or the CPU-wide settings.
- A customer may choose to give the IPUCOM program to a SUPER group member and then set PROGID on it. They can then use either Safeguard ACLs or standard file security to control execution access. It is not shipped with PROGID set.

For more information about the restrictions and requirements on the use of IPUCOM, see the considerations for the IPUAFFINITY_GET_, _SET_, and _CONTROL_ procedures in the *Guardian Procedure Calls Reference Manual.*

Examples

The following example sets the primary $XYZ process on IPU 1:

```
IPUCOM -pname $xyz 1
```

The following unbinds the primary $XYZ process, allowing it to be moved again by the process scheduler:

```
IPUCOM -pname $xyz unbind
```

The following displays the IPU affinity of the $XYZ process:

```
IPUCOM -pname $xyz info
```

The following sets the process that is PIN 456 in CPU 3 on IPU 2:

```
IPUCOM -pin 456 -cpu 3 2
```

The following turns off DP2 balancing in the process scheduler in CPU 5:

```
IPUCOM -cpu 5 -balanceDP2 off
```

JOIN Command

Use the JOIN command to convert a multiple-line variable level into a single-line variable level, replacing each internal end-of-line with a single space.

```
JOIN variable-level
```

variable-level

is the name of an existing variable level.

KEEP Command

Use the KEEP command to remove all but the top num definition levels of one or more variables.

```
KEEP [ / LIST / ] num variable [ variable ] ...
```

LIST

causes KEEP to display each variable name followed by the number of levels removed from that variable.

num

is the number of levels to keep.

variable

is the name of an existing variable.

Consideration

KEEP 0 deletes the variable.

KEYS Command

Use the KEYS command to display the current function key definitions.

```
KEYS
```

Example

This example is a sample KEYS command display; the values vary depending on how the TACL environment is set up:

```
127> KEYS
F16   = (The Help Key)
F1    = #OUTPUT TEDIT %1%; %2 to *%
        TEDIT %1%; %2 to *%
        #OUTPUT Finished editing [#SHIFTSTRING/UP/%1%]
F2 =    #OUTPUT TFORM /IN %1%, OUT $S.#%2%, NOWAIT/ %3 to *%
        TFORM /IN %1%, OUT $S.#%2%, NOWAIT/ %3 to *%
F3 =    TYPE %1%

  ***

SF15 =  #OUTPUT VOLUME %1%
        VOLUME %1%
SF16 =  #OUTPUT LOGOFF
        [#IF NOT [#MATCH $DECAY [#MYTERM]] |THEN|
          LOGOFF /SEGRELEASE/
        |ELSE|
        LOGOFF
]
```

LIGHTS Program (Super-Group Only)

Use the LIGHTS program to control the processor panel lights (also known as the switch-register display). You must use a super-group user ID (255,*your-id*) to issue this command.

This command is not supported on NonStop S-series systems after the G06.04 RVU.

```
LIGHTS [ ON | OFF | SMOOTH { num }]
  [  [ , sys-option ] | [, ALL ] ]
  [ , BEAT ]
```

ON

enables the display of processor usage in the processor panel lights. ON is the default if you do not specify any of these parameters in the command that runs the LIGHTS program.

OFF

stops the lights display and turns off all the processor panel lights.

SMOOTH [*num*]

reduces the variance in the processor usage display. It causes the lights to show processor usage over the previous num seconds, where *num* is a value between 1 and 60, inclusive. The default value of *num* is 10. Although the processor usage is averaged over num seconds, the lights display is refreshed every second. LIGHTS SMOOTH also turns on the processor lights if they are off.

You can include a sys-option, and the ALL and BEAT parameters, only when the display is enabled (that is, when LIGHTS ON or LIGHTS SMOOTH is in effect).

sys-option

is any of these three:

DISPATCHES

flashes processor light 13 to indicate passage of 100 dispatches. (After 95 dispatches, light 13 is lit for 5 dispatches.)

SYSPROCS

turns on processor light 14 whenever a system process is running.

PAGING

turns on processor light 15 whenever a page fault is being processed (that is, the memory manager is waiting for a page transfer from a disk process).

ALL

turns on every *sys-option* at once (DISPATCHES, SYSPROCS, and PAGING).

BEAT

flashes light 0 (zero) once every second to indicate that the processor is functioning.

Considerations

- To run the LIGHTS program, you must have a group ID of 255.
- Processor lights are lit for these reasons:
  - Lights 0 through 10 indicate processor usage. Light 0 is always lit (or flashing if BEAT was specified). One additional light is lit for each 10 percent increase in processor usage.
  - Processor light 11 is always off.
  - Light 12 is lit if the processor successfully recovered from a power failure.When you run the LIGHTS program, light 12 is turned off in all processors.
  - Lights 13 through 15 are lit for the reasons given in the syntax description for DISPATCHES, SYSPROCS, and PAGING, respectively. Each light is lit for a larger percentage of the time if more processing time is spent on the corresponding activity.
- When the system is cold loaded, the default LIGHTS setting is ON, ALL, BEAT.

Examples

1. To turn on the processor lights and have them show processor usage smoothed over 10-second intervals, enter:

```
94> LIGHTS SMOOTH
```

2. To change the display to show processor usage for 25-second intervals, enter:

```
95> LIGHTS SMOOTH 25
```

3. To turn on processor light 15 whenever paging activity occurs, enter:

```
96> LIGHTS ON, PAGING
```

4. To turn on light 14 whenever a system process is executed, and light 13 after each 100 dispatches, enter:

```
97> LIGHTS, SYSPROCS, DISPATCHES
```

LOAD Command

Use the LOAD command to load one or more TACL library files into the TACL memory.

```
LOAD [ / KEEP num / ] file-name [ file-name ] ...
```

KEEP *num*

causes TACL, after executing the LOAD command, to perform an implicit KEEP command on all the variables modified by the LOAD command.

file-name

is the name of one or more TACL libraries.

Considerations

- Each library consists of one or more ?SECTION directives that specify a variable and a type associated with that variable, followed by the information (body) to be put into it when it is created.
- To process a file, TACL does this for each ?SECTION directive: it pushes the variable, sets its contents to type, and makes body the new top-level definition of the variable.
- As a library is loaded, comments are removed to conserve variable space. Any line that is blank, or becomes blank because of comment removal, is discarded.
- If you need to include a blank line (often useful in DELTA type variables), use the ?BLANK directive. ?BLANK causes the loader to insert a blank line in the variable level.
- To include lines beginning with question marks (for example, you might be loading DDL commands into a variable level for later use as the IN variable for a DDL run), double the question mark (??). The first question mark and any spaces adjacent to it are discarded and the remainder of the line is treated as text.
- The LOAD command reads data from a library file in TACL format unless the file contains a ?FORMAT PLAIN or ?FORMAT QUOTED directive to specify otherwise. If changed, the format reverts to TACL at the next ?SECTION directive.

Example

This example illustrates how to load two libraries, retaining only one level of each variable:

```
17> LOAD / KEEP 1 / mykeys mymacs
```

LOADEDFILES Command

Use the LOADEDFILES command to display information about the load files that are being used by the specified process.

```
LOADEDFILES [ / OUT list-file / ][ \node-name ] &
            { $Process-name | CPU, PIN|}
```

`OUT` *`list-file`*

specifies a device or sequential file accessible to the sequential I/O (SIO) facility that is to receive the output from the command. If you omit this option, TACL writes the listing to the current OUT file.

If you specify an OUT file that does not exist, TACL creates an EDIT file named list-file. If you specify an OUT file that already exists, TACL appends the information to the end of the file.

`\node-name`

is the name of the system on which the specified process executes.

`$Process-name` `or` *`CPU, PIN`*

is the name of the process or process pair. Alternatively, use the CPU number and process identification number of the process.

This information is displayed:

Filename - the fully qualified LOADEDFILE name

If the specified process is a Guardian file, the files are reported using a Guardian format, for example

```
\BOMBAY.$SYSTEM.SYS01.ZCRESSRL
```

If the specified process is OSS then path names are reported using the OSS format, for example

```
OSS file: /bin/ksh
Guardian file: /G/system/sys01/zcresrl
```

Type - the type of loaded file; it can be any one of these types:

N-PROG: a native program (non-PIC).
P-PROG: a native program (PIC)
UC-R: a TNS program (accelerated)
UC: a TNS program (non-accelerated)
UL-R: a TNS User Library (accelerated)
UL: a TNS User Library (accelerated)

SRL: a Shared Run-time Library (including a non-PIC native UL)
DLL: a Dynamic-Link Library (including a PIC native UL)
ERR: Loadedfile unloaded in between execution of program
ERROR: Unrecognized loadfile type

Considerations

- If you do not specify a process ($process-name or CPU, PIN), LOADEDFILES lists all the loaded files for the current TACL processes.
- If you omit the node name, the default system is assumed.
- If you omit the process name by using CPU, PIN method then the process name will be displayed as UNNAMED PROCESS.
- If the process is an unnamed process, the process name will be displayed as UNNAMED PROCESS.
- The LOADEDFILES command lists information for only one process at a time.

Example

```
$DATA08 KIRAND 10> loadedfiles \bombay.0,11

PROCESS NAME =     $ZL00
TYPE      FILENAME
NPROG     \BOMBAY.$SYSTEM.SYS01.ROUT
SRL       \BOMBAY.$SYSTEM.SYS01.ZCRESRL
SRL       \BOMBAY.$SYSTEM.SYS01.ZI18NSRL
SRL       \BOMBAY.$SYSTEM.SYS01.ZICNVSRL
SRL       \BOMBAY.$SYSTEM.SYS01.ZCRTLSRL
SRL       \BOMBAY.$SYSTEM.SYS01.ZCPLGSRL
SRL       \BOMBAY.$SYSTEM.SYS01.ZOSSKSRL
SRL       \BOMBAY.$SYSTEM.SYS01.ZTLHGSRL
Total No. of Loadedfiles = 8
```

LOGOFF Command

Use the LOGOFF command to log off from the current TACL process. The LOGOFF command is an alias for the #LOGOFF Built-In Function on page 9-252.

```
LOGOFF [ / option [, option ] ... / ]
```

option

is one of these options:

CLEAR

clears the terminal screen after you log off (for use when TACL is not configured to clear the screen automatically).

NOCLEAR

prevents TACL from clearing the terminal screen after you log off (for use when TACL is configured to clear the screen automatically, as is usually the case).

PAUSE

causes TACL to execute a PAUSE command immediately following the LOGOFF command.

SEGRELEASE

causes TACL to release immediately the extended segment that held your variables. Typically, TACL saves this segment in the event you later log onto the same TACL; but if you fill up your variable space and processing cannot proceed, you can log off using the SEGRELEASE option to discard the variable space.

Considerations

- When you log off, you terminate only your session with the current TACL.Processes that you started can continue to run after you log off.
- Any macro or routine running at the time #LOGOFF occurs terminates immediately. A subsequent LOGON does not restart it.
- Before logging off, you should stop processes that you are no longer using. Use the STATUS Command on page 8-206 to see a list of the processes that are running. Then use the STOP Command on page 8-215 to stop any unneeded processes.
- If you enter the LOGOFF command while working through a modem, the modem disconnects (unless the ancestor of your TACL process is running in another system, as above).

- If the ancestor of your current TACL process is a process running in another system and you enter the LOGOFF command, the current TACL process is deleted and control returns to the ancestor process. This message is displayed:

  ```
  Exiting from TACL on \node-name
  ```

- If you are accessing a remote node through a modem on your local node, TACL does not issue a modem disconnect.
- After you log off, the current TACL process accessor ID and creator accessor ID are set to NULL.NULL (0,0). This is also the setting of the accessor IDs when an interactive TACL is started but no user has logged on.
- After you log off, any process that tries to use your variables receives file-system error 66 on its I/O requests.

Note. The LOGOFF command is an alias for the #LOGOFF built-in function. You can, if you want, redefine LOGOFF as the name of a routine or macro of your own creation that checks for variables in use by processes, requesters, and servers. You can then close associated files and processes before logging off to avoid error-66 conditions.

- If your TACL is interactive, and your terminal is a 6520 or 6530, and TACL is configured to clear the terminal screen as a security measure (the default), it does so when you log off. You can override the automatic screen clearing with the NOCLEAR option. Conversely, if TACL is configured to omit automatic screen clearing, you can use the CLEAR option to clear the terminal screen.

LOGON Command

Use the LOGON command to establish communication with a TACL process. To enter TACL commands interactively, you must first log on. However, when you execute TACL commands from a command file (by specifying a disk file containing TACL commands as the IN file when you start a new TACL process), you do not need to include the LOGON command in the file.

Note. The LOGON command behavior depends on the Safeguard environment. If Safeguard is not running on your system or if the USER_AUTHENTICATE procedure is not in the system library, these logon options are not available:

- *alias*
- *old-password, new-password*
- *old-password, new-password, new-password.*

```
LOGON
  group-name.user-name | group-id,user- id | alias
  [  , password |
     , old-password, new-password |
     , old-password, new-password, new-password
  ]
  [  ; parameter  [ , parameter ]  ]
```

group-name.*user-name* or *group-id*,*user-id*

is the group name and user name, or the group number and user number, of the user who is logging on. The user identity must already be established. If you do not have a user account, see your system administrator. To do a blind logon, enter only LOGON followed by your *group-name.user-name* or *group-id, user-id* and press RETURN. TACL then prompts you for your password.

If the TACL configuration NAMELOGON option is not set to or if Safeguard is running and the Safeguard NAMELOGON flag is set to 0 (for USER_AUTHENTICATE_ call), the group number and user number is not accepted.

alias

is an alternate assigned name. Each alias must be unique within the local system. An alias is a case-sensitive text string that can be up to 32 alphanumeric characters in length. In addition to alphabetic and numeric characters, the characters period (.), hyphen (-), and underscore(_) are permitted within the text string. The first character of an alias must be alphabetic or numeric. For more information on aliases, see the *Safeguard Reference Manual*.

password

is the password associated with the user. You must include password if a password has been established for you. If you have a password, you must separate it from your user name with a comma if you enter it before pressing RETURN. You can also use the blind password feature by omitting the comma and password from the LOGON command, then entering your password when prompted. For more information about passwords, see the PASSWORD Program on page 8-115.

You can change your password as part of the logon sequence. Initially, the logon dialog is the same as a normal logon. However, to indicate that you want to change your password, type a comma at the end of your password. The system prompts you for a new password and then requests reentry of the new password to verify it. This dialog shows a sequence in which SUPPORT.JANE changes her password from alpha4 to BigTop:

```
TACL 1> LOGON SUPPORT.JANE
Password: alpha4,
Enter new password: BigTop
Reenter new password: BigTop
The password for SUPPORT.JANE has been changed.
```

An alternative method of changing a password is to enter the current password, the new password, and the verification of the new password on the same line. This dialog shows this type of password change:

```
TACL 1> LOGON SUPPORT.JANE
Password alpha4, BigTop, BigTop
The password for SUPPORT.JANE has been changed.
```

Another option for changing the password is to enter the current and new passwords on one line and the verification of the new password on the new line.

```
TACL 1> LOGON SUPPORT.JANE
Password alpha4, BigTop
Reenter new password: BigTop
The password for SUPPORT.JANE has been changed.
```

parameter

is an operating parameter for the TACL process. It can be one of these:

```
ABENDONABEND
HOMETERM
SEGVOL $ volume-name
STOPONABEND
```

Note. Parameters can be specified only when you are logging on from the logged-off state. You cannot specify parameters when you are logging on from the logged-on state.

ABENDONABEND

specifies that TACL abends with an abend completion code and displays an error message to the current OUT file when a process started by this TACL

abends or stops with an abnormal completion code. If, however, the subordinate process was started in the NOWAIT mode or with any JOBID value, including 0, the parent TACL process does not stop. In such a case, TACL displays the error message but continues to run.

If the CPU of a single process fails, the condition is treated as if the process abended.

If you start a process pair from this TACL process and the CPU of one of the processes fails, this has no effect on the TACL process; both processes in the pair must cease to exist before the ABENDONABEND option applies.

HOMETERM

specifies that TACL is to use the device specified by the TERM run-option as the home terminal. If you omit HOMETERM, if the TACL IN file is the same as the TACL OUT file and TACL is not in server mode, TACL uses its IN file device as the home terminal, regardless of any specification by the TERM option. If the IN file is the same as the OUT file and the TACL process is not named, TACL does not set its home terminal.

A process started by TACL inherits its home terminal unless the RUN command that initiates the process specifies a different home terminal.

SEGVOL $*volume-name*

can be included only on a “cold” logon (when entering from the logged-off state); it indicates the name of the disk volume where the extended segment swap file resides.

STOPONABEND

specifies that TACL abends and displays an error message to the current OUT file when a process started by this TACL abends or stops with an abnormal completion code. (If, however, the subordinate process was started in the NOWAIT mode or with any JOBID value, including 0, the parent TACL process does not stop. In such a case, TACL displays the error message but continues to run.)

If the CPU of a single process fails, the condition is treated as if the process abended. If you start a process pair from this TACL process and the CPU of one of the processes fails, this has no effect on the TACL process; both processes in the pair must cease to exist before the STOPONABEND option applies. For a description of completion codes, see the *Guardian Procedure Calls Reference Manual*.

Considerations

- When you log on, the operating system displays a logon message that usually includes:

  - The operating system RVU number and date
  - The number of the processor in which the primary and backup TACL process is running
  - The current system date and time

- When you enter a full LOGON command, your identity and password are displayed on the terminal screen; if you want to assure security, you can use the blind password feature.
- When you enter the LOGON command, your logon (saved) default values are in effect for system, volume, and subvolume names and for disk file security. (See the DEFAULT Program on page 8-53 for information about logon default values.)
- After you log on, the current TACL process accessor ID and creator accessor ID are set to the accessor ID associated with your user name. If your logon fails because you try an invalid or undefined user name or password, the accessor IDs remain set at 0,0. (This is the accessor ID setting when an interactive TACL is started and no user has logged on.)
- A logon failure occurs if either the $CMON process or the USER_AUTHENTICATE_ procedure denies the logon but not if you make a syntax error in the LOGON command. If a logon failure occurs, TACL displays an “Invalid username or password” message, without specifying which element was in error. If successive logon failures occur, TACL can prevent further logons for a specific period of time, depending on Safeguard and USER_AUTHENTICATE_ security logic. Until a successful logon occurs, all subsequent logon failures also initiate this delay.
- If the USER_AUTHENTICATE_ procedure fails to recognize the user information given during the logon operation, the TACL built-in variable #ERRORNUMBERS contains the error information. To display the error information, issue these commands:

  ```
  #PUSH n1 n2 n3 n4
  #SETMANY n1 n2 n3 n4, [#ERRORNUMBERS]
  ```

  where:

  n1 = 1074 (Invalid user name or password)
  n2 = error return from USER_AUTHENTICATE_
  n3 = error return detail from USER_AUTHENTICATE_
  n4 = 0

  If *n2* contains 0 (no error state returned), USER_AUTHENTICATE_ is not in the system library or Safeguard is not running. If Safeguard is not running, *n3* and *n4* are set to 0.

- If the USER_AUTHENTICATE_ procedure exists in the system library, TACL calls the USER_AUTHENTICATE_ procedure, a procedure that uses the Safeguard facility, to verify users logging on. Otherwise, TACL calls the VERIFYUSER system procedure.

- When you enter a full LOGON command, your identity and password are displayed on the terminal screen. You can use the blind password feature in the TACL configuration or in Safeguard to ensure security.
- The process accessor ID and creator accessor ID of the user logging on are propagated to any descendant processes of TACL. (For a description of accessor IDs, see the *Guardian User's Guide*.)
- If you are logged on as the super ID, you can log on as any user in your system without giving that user's password. Similarly, if you are a group manager, you can log on as any member of your group without giving that user's password.
- Entering the LOGON command from a logged-off state clears all assignments you made with the ASSIGN and PARAM commands, clears all DEFINEs, and resets the Inspect setting to the default value of OFF (see the SET INSPECT Command on page 8-193 for information about Inspect settings). But if you log on without explicitly logging off the previous user, the assignments, parameters, and DEFINEs are retained, and the current Inspect setting remains in effect (see Example 3 on page 8-106).
- The ability to log on from a logged-on state can be disabled by setting the TACL configuration option NOCHANGEUSER to -1. To display the value of NOCHANGEUSER, use #GETCONFIGURATION /NOCHANGEUSER/.
- If the TACL configuration option BLINDLOGON is not set to 0 or if Safeguard is running and the Safeguard NAMELOGON flag is not set to 0 (for USER_AUTHENTICATE_ call), the use of the comma is prohibited. The password must be entered at its own prompt while echoing is disabled. To display the value of BLINDLOGON, use #GETCONFIGURATION /BLINDLOGON/.
- If the TACL configuration option REMOTECMONREQUIRED is not set to 0, all operation requiring approval by remote $CMON are denied if that remote $CMON is unavailable or is running too slowly. To display the value of REMOTECMONREQUIRED, use #GETCONFIGURATION /REMOTECMONREQUIRED/.
- The configuration information in effect for a particular user ID depend on the setting of the REQUESTCMONUSERCONFIG option.

  When a TACL process is started, the configuration information consists of default values. The process is in the no-user-logged-on state.

  If a TACL process is in the no-user-logged-on state when a LOGON command is received and REQUESTCMONUSERCONFIG is OFF, the TACL process sends an initial request to the CMON process for configuration information and then sends the LOGON request.

  - If the CMON process does not return configuration information, then the TACL process uses the configuration information it had for the no-user-logged-on state.

- If the CMON process does return the configuration information, then the TACL process uses this information.

If the TACL process is in the no-user-logged-on state when a LOGON command is received and REQUESTCMONUSERCONFIG is ON, the TACL process sends a request to the CMON process for configuration information, then sends the LOGON request, and then sends an additional request for the configuration information.

- If the CMON process does not return the configuration information, then the TACL process uses the information it had for the no-user-logged-on state.
- If the CMON process does return the configuration information, then the TACL process uses this information. The CMON process has had the opportunity to change the configuration information that it wants associated with the user ID.

If a TACL process is in the user-logged-on-already state when a LOGON command is received and REQUESTCMONUSERCONFIG is OFF, the TACL process authenticates the user ID, and then sends the LOGON request.

- The TACL process uses the configuration information it was using for the previous user ID.

If the TACL process is in the user-logged-on-already state when a LOGON command is received and REQUESTCMONUSERCONFIG is ON, the TACL process authenticates the user ID, then sends the LOGON request, and then sends a request for the configuration information.

- If the CMON process does not return configuration information, the TACL process uses the configuration information it was using for the previous user ID.
- If the CMON process does return information, then the TACL process uses this information. The CMON process has had the opportunity to change the configuration information that it wants associated with the user ID.

- ABENDONABEND and STOPONABEND: If you logon with both the STOPONABEND and ABENDONABEND parameters, the last parameter specified in the list overrides the earlier ones specified.

  In this case, STOPONABEND overrides ABENDONABEND:

  ```
  12> TACL /NAME/ ;ABENDONABEND, STOPONABEND
  ```

  The STOPONABEND or ABENDONABEND parameter specified at TACL startup is the default setting for all TACL logon sessions started from that TACL. You can override the default setting by specifying a different parameter in the LOGON command.

In this case, ABENDONABEND overrides the default setting (STOPONABEND) just for this logon session:

```
12> TACL /NAME/ ;STOPONABEND
TACL 1> LOGON SOFTWARE.JANE ;ABENDONABEND
Password:
```

When TACL abends as result of the ABENDONABEND being specified at TACL startup or at logon, and a process started by this TACL abends or stops with an abnormal completion code, this abend message is displayed.

```
*ABEND*
*ERROR* TACL stopped by a process ABEND/STOP - PID:
ABENDED: $MP
```

The TACL process abend completion code will be returned in the :_COMPLETION and :_COMPLETION^PROCDEATH variables.

```
29> OUTVAR :_COMPLETION
_COMPLETION(0)
     MESSAGECODE(0:0
                 -6
     PROCESS(0:0)    $MP
     HEADERSIZE(0:0) 0
     CPUTIME(0:0)    1854774
     JOBID(0:0)      0
     COMPLETIONCODE(0:0)
                   5
     INTERNAL(0)
          TERMINATIONINFO(0:0)
                    0
          SUBSYSTEM(0:0)
     TEXTLENGTH(0:0) 0
     TEXT(0:79)

30> OUTVAR :_COMPLETION^PROCDEATH
 _COMPLETION^PROCDEATH(0)
      Z^MSGNUMBER(0:0)
                   -101
      Z^PHANDLE(0:0)  512.675.3.152.0.0.574.34458.0.175
      Z^CPUTIME(0:0)  1854774
      Z^JOBID(0:0)    0
      Z^COMPLETION^CODE(0:0)
                    5
      Z^TERMINATION^CODE(0:0)
                    0
      Z^SUBSYSTEM(0:0)
      Z^KILLER(0:0)   65535.65535.65535.65535.65535...
      Z^TERMTEXT^LEN(0:0)
                   0

      Z^PROCNAME(0)
           ZOFFSET(0:0)    82
           ZLEN(0:0)       20
       Z^FLAGS(0:0)    1
       Z^RESERVED(0:2) 1 1 0
```

```
Z^DATA(0)
     BYTE(0:111)     \PRUNE.$MP:150608489
```

Examples

1. User SOFTWARE.JANE, whose password is STAR, can log on interactively by entering:

   ```
   TACL 1> LOGON SOFTWARE.JANE,STAR
   ```

2. Using the blind logon feature, SOFTWARE.JANE can log on in this sequence:

   ```
   TACL 1> LOGON SOFTWARE.JANE
   Password:
   ```

   At the “Password:” prompt, she enters her password, which does not appear on the screen. TACL displays a logon message, followed by its prompt, consisting of a history number, a greater-than symbol, and a space (1>). SOFTWARE.JANE can now enter her next command.

3. This example shows how to log on as another user and retain all your current settings. The user MANUF.FRED has logged on and is using the system. Next he logs on with the user name MANUF.MABEL (he must know her password); this LOGON command implicitly logs FRED off. He changes the security of one of Mabel’s files, and then he logs on again with his own ID. The final logon also logs off MANUF.MABEL:

   ```
   7> LOGON MANUF.MABEL
   Password:
   8> FUP SECURE BIG.FILE, "NNNU"
   9> LOGON MANUF.FRED
   Password:
   ```

   All of Fred’s logon defaults remain unchanged, as well as any settings he may have made with ASSIGN, PARAM, and SET commands.

_LONGEST Function

Use the _LONGEST function to obtain the length of the longest element in a list of elements.

```
_LONGEST list
```

list

is a list of elements separated by spaces, or a variable level containing such a list.

Result

_LONGEST returns the length of the longest element supplied in its argument.

Example

This example illustrates the use of _LONGEST to determine the length of the longest item in a variable level:

```
12> PUSH do
13> SET VARIABLE do GO RUN EXECUTE
14> #OUTPUT [_LONGEST [do] ]
7
```

_MONTH3 Function

Use the _MONTH3 function to obtain the three-letter abbreviation for month num.

```
_MONTH3 num
```

num

is a one- or two-digit number indicating a month.

Result

_MONTH3 returns the three-letter abbreviation for the specified month.

Example

This example illustrates the use of _MONTH3 to convert the numeric representation of a month to a three-letter textual representation:

```
13> #OUTPUT [_MONTH3 3]
MAR
```

O[BEY] Command

Use the OBEY command to execute a series of TACL commands or built-in functions contained in a file.

```
O[BEY] command-file
```

command-file

is either the name of an existing disk file that contains TACL commands or built-in functions, or the name of a running process.

Considerations

- If command-file is the name of an existing file, TACL opens the file in read-only mode and interprets each logical line in the file as a command or command sequence to be executed.
- If command-file is the name of an existing process, TACL performs a WRITEREAD operation, sending the process a prompt of the form “OBEY> ” and waiting for the process to reply. The response, presumably a command or command sequence for TACL to execute, can contain as many as 239 characters. TACL continues to prompt and accept replies until the process sends back an end-of- file character or terminates.
- The OBEY command:
  - Echoes the line
  - Interprets the statement, including metacharacters
- TACL does not check the file type or contents before trying to interpret the file. If the file does not contain TACL commands, the TACL process tries to interpret each line and returns errors.
- You cannot include TACL directives (such as ?TACL or ?FORMAT) in an OBEY command file. For example, the OBEY command returns an error when interpreting a file that starts with a ?TACL MACRO directive.
- You can stop the execution of commands in an OBEY file by pressing the BREAK key at the terminal where you entered the OBEY command. TACL closes the OBEY file and prompts you for the next command. (Note, however, that you can set the #BREAKMODE built-in variable to disable the BREAK key; in this case, TACL ignores the BREAK key.)

Example

To execute the commands in the file ALLFILES (assuming that ALLFILES is in the current subvolume), type:

```
14> OBEY allfiles
```

OUTVAR Command

Use the OUTVAR command to display the contents of a variable without executing or invoking it. The OUTVAR command is an alias for the #OUTPUTV built-in function.

```
OUTVAR [ / option [ , option ] ... / ] string
```

option

is an option that qualifies the write operation. Options are not permitted if variable is of type STRUCT. Specify option as one of these:

`COLUMN` *num*

begins writing data at column num. If text in the buffer already extends beyond the specified column, TACL outputs the buffer first and starts a new line.

`FILL { SPACE | ZERO }`

specifies the character (space or zero) that is to fill the unused character positions to the right if the output is narrower than the number of columns specified by WIDTH.

`HOLD`

holds the last line of the variable output in a buffer until one of these events occurs:

- OUTVAR, #OUTPUT, or #OUTPUTV is executed without the HOLD option
- The output buffer becomes full
- TACL, #DELTA, #INPUT, or #INPUTV prompts for input
- #DELTA exits

`JUSTIFY { LEFT | CENTER | RIGHT }`

specifies, if the output is less than the number of columns specified by WIDTH, whether the output is to be left-justified, right-justified, or centered. If the output is equal to or greater than the WIDTH specification, JUSTIFY is ignored.

`WIDTH` *num*

specifies the width of the field into which the output is to be placed. If omitted, WIDTH defaults to the actual width of the text to be output.

`WORDS`

specifies that each line is to be treated as a space-separated list and that the FILL, JUSTIFY, and WIDTH options are to be applied to the individual members of the list. If this option is omitted, each line is treated as a single output item.

string

is the data to be output. It is the name of an existing variable level, text enclosed in quotation marks, or a concatenation of such entities. The concatenation operator is '+' (the apostrophes are required).

Considerations

- If options are supplied, the options are applied to each line of the variable as though you had called the #OUTPUT built-in with the same options for each line of the variable. In particular, the last line of a variable is the only one that the HOLD option really applies to because each line forces output of any previous line.
- OUTVAR with the HOLD option is useful for constructing lines to be output piece by piece. For example, when you invoke this macro file, called BUNCHUP:

```
?TACL MACRO
#PUSH part
#SET part All
OUTVAR /HOLD/ part
#SET part together
OUTVAR /HOLD/ part
#SET part now
OUTVAR part
```

  the text all comes out on one line, as follows:

```
13> BUNCHUP
Alltogethernow
```

- No options are allowed when a structure or substructure is being displayed.
- OUTVAR structure is not the same as #OUTPUT [structure]. The former displays the data of the structure in a stylized format, such as:

```
14> OUTVAR inventory
INVENTORY(0)
  ITEM(0:0) 123
  PRICE(0:0) 1004
  QUANTITY(0:0) 97
```

  while the latter merely displays the data as a space-separated list of its external representations.

- OUTVAR does not show redefinitions unless the argument itself is one.
- If #OUTFORMAT is set to PRETTY, you can control output spacing by the use of tilde-underscore (~_) metacharacters in the data to be displayed by OUTVAR.

Example

This example illustrates the use of OUTVAR to display the contents of a variable. First, define a variable name vara:

```
15> [#DEF vara MACRO |BODY|
15> #OUTPUT [#FILEINFO / EXISTENCE / tempfile ]
15> ]
```

At this point, if you type vara, TACL returns 0 (false) as the result of vara if TEMPFILE does not exist and -1 (true) if it does exist.

```
16> vara
0
```

If you now enter the OUTVAR command for the variable vara, you see the actual contents of vara:

```
17> OUTVAR vara
#OUTPUT [#FILEINFO / EXISTENCE / tempfile ]
```

PARAM Command

Use the PARAM command to create a parameter and give it a value or to display all current parameters and their values.

```
PARAM [ param-name param-value
  [ , param-name param-value ] ... ]
```

param-name

is a user-defined parameter name to be assigned a value. param-name can consist of 1 to 31 alphanumeric, circumflex (^), and/or hyphen (-) characters.

param-value

is the value assigned to param-name.

Considerations

- Entering PARAM with no arguments displays the names and values of all currently defined parameters.
- Comments and leading and trailing spaces are deleted in param-value.
- TACL stores the values of parameters assigned by the PARAM command and sends the values to processes that request parameter values when the processes are started. The interpretation of parameter values is made by the processes that request them.
- To delete existing parameters, use the CLEAR command.
- All parameters are deleted when you use the LOGOFF command. Parameters and their values are retained, however, if you enter a LOGON command without logging off first.
- If you start a new TACL process from your existing TACL process, the new TACL process does not inherit existing PARAM values.
- When a backup TACL process takes over, TACL clears all PARAMs.
- TACL reserves 1024 bytes of internal storage for parameters and their values. The number and length of parameters in effect are limited by this storage area.
- From a TACL macro or routine, use #PARAM to display a list of all parameters or the value of a specified parameter.
- The same set of PARAM attributes can be configured for a generic process through SCF. For the syntax, see the *SCF Reference Manual for the Kernel Subsystem*.

Example

To assign the value ON to the parameter TRACE:

```
19> PARAM TRACE ON
```

PASSWORD Program

Use the PASSWORD program to create, change, or delete your password. (Passwords are optional and might not be required on your system.)

```
PASSWORD [ / run-option [ , run-option ] ... / ][ password ]
```

run-option

is any of the options described in the RUN[D|V] Command on page 156.

password

is the new password that you must enter when you log on. If you omit password, your current password is deleted and no password is required to log on with your user name. A password can include from one to eight characters, except for spaces, commas, and null characters. Lowercase letters are not converted to uppercase. Do not end the password with an ampersand (&) character unless you intend to continue the password on the next line.

Considerations

- PASSWORD entered with no following parameters deletes your current password. Thereafter, no password is required to log on with your user name.
- A new user who has just been added to the system has no password.
- Although you can use control characters in your password, entering control characters might cause undesired changes in terminal operation. On some terminals, certain control characters cannot be used in passwords. Refer to the user documentation for your terminal for information about the effect of control characters on terminal operation.

Example

If you are user 8,44, you can give yourself the password “schubert” (all lowercase) by entering:

```
14> PASSWORD schubert
THE PASSWORD FOR USER (008,044) HAS BEEN CHANGED.
```

You must now enter this password whenever you log on.

PAUSE Command

You can run more one or more processes simultaneously from within your TACL process. Use the PAUSE command to cause TACL to wait for prompting from-or completion of-another process.

```
PAUSE [ [\node-name.]{$process-name | cpu,pin } ]
```

\node-name

is the name of the system where the process identified by $*process-name* or *cpu,pin* is running. If you omit *\node-name*, the current default system is used.

$process-name

is the name of a process for which TACL is to pause. TACL does not prompt for commands until it receives a process deletion message from that process.

cpu,pin

is the CPU number and process identification number of a process for which TACL is to pause. TACL does not prompt for commands until it receives a process deletion message from that process.

Considerations

- PAUSE with no parameters causes TACL to pause for the last process started from the current TACL, or the process for which TACL last paused, if that process is still running.
- If the process is interactive, you can exit the process and regain control of your terminal by pressing the BREAK key.
- When you enter a PAUSE command, the current TACL process stops prompting for commands, allowing the specified process (or the default process) to gain control of your terminal if it needs to use it for communication. When the other process finishes, or is deleted, the operating system sends a process deletion message to TACL. When TACL regains control of your terminal, it resumes prompting.

  Assume, for example, that another process controls your terminal, and you press the BREAK key to regain control of the terminal. To return control to that other process, enter PAUSE. TACL regains control of your terminal after that other process finishes or is deleted.
- To list the processes that are currently running on your terminal, use the STATUS *, TERM command. If a process has a name, you can include its *cpu,pin* in a PPD command and obtain its $*process-name*.
- You can use the WAKEUP command to specify the action TACL takes when it receives a process deletion message.

- If WAKEUP is set to ON, TACL regains control of the terminal when any process started from it is deleted.
- If WAKEUP is set to OFF (the default setting), TACL regains control of the terminal when the process specified in the PAUSE command is deleted. If no process was specified in the PAUSE command, TACL regains control of the terminal when the last process started from it is deleted.

- If you enter PAUSE with no parameters when the most recent process started from the current TACL has already been deleted, or if you specify a process that the current TACL did not create, TACL waits forever (no prompt appears on the screen) unless you press the BREAK key or TACL receives a wake message, indicating that a process was deleted.
- You can enter the PAUSE command, without any parameters, even when you are logged off.

Examples

This example illustrates how to use the BREAK key to temporarily leave a process, such as EDIT, and return control of your terminal to TACL. After checking the status of the EDIT process, the user returns to the EDIT process and terminates the EDIT process:

```
11>EDIT FILE2
TEXT EDITOR - T9601B30 - (08MAR87)
*
```

(BREAK pressed)

```
12> STATUS *,TERM
Process Pri PFR %WT Userid Program file Hometerm
5,49 120 R 000 8,230 $SYSTEM.SYS01.TACL $TE0.#A
5,55 119 000 8,230 $SYSTEM.SYSTEM.EDIT $TE0.#A

13> PAUSE
*EXIT

14> !STA
14> STATUS *,TERM
Process Pri PFR %WT Userid Program file Hometerm
5,49 120 R 000 8,230 $SYSTEM.SYS01.TACL $TE0.#A
```

The asterisk prompt (*) indicates that EDIT has control of the terminal. Pressing the BREAK key returns control of the terminal to TACL. When you enter PAUSE, the EDIT prompt reappears.

PMSEARCH Command

Use the PMSEARCH command to define the list of subvolumes that TACL is to search to find program and macro files in response to an implied RUN command.

```
PMSEARCH subvol-spec [ [,] subvol-spec ] ...
```

subvol-spec

specifies a subvolume to be searched. subvol-spec takes one of these forms:

[*node-name*.][$*volume*.] *subvol*

is a syntactically correct name of an existing subvolume. If you omit *node-name* or $*volume*, the current default system or volume is assumed.

#DEFAULTS

is a TACL keyword that, in this specific context, represents the default volume and subvolume in use at the time you issue an implied RUN command.

Considerations

- If $SYSTEM.SYSTEM appears in a search list and TACL searches it but fails to find the desired file, TACL automatically searches the current $SYSTEM.SYSnn subvolume as well.
- If you do not define a program and macro search list, TACL searches only in $SYSTEM.SYSTEM and $SYSTEM.SYS *nn* to find a file specified in an implied RUN command.
- If TACL finds the desired file but its security does not allow TACL to invoke it, TACL stops at that point in the search list.
- When you issue an implied RUN command, TACL searches the first subvolume specified in the search list. If it fails to find the program or macro file there, it searches the next subvolume, and so on. After it finds the program or macro file, the search ends. For example:

```
27> VOLUME $OLD.HOME
28> PMSEARCH $SYSTEM.SYSTEM #DEFAULTS [#DEFAULTS]
29> VOLUME $NEW.PLACE
30> EDIT FRED
```

#DEFAULTS is a TACL keyword that, in this specific context, represents the default volume and subvolume. [#DEFAULTS] represents the defaults at the time the search list is created; #DEFAULTS represents the defaults at the time the search list is accessed to find a program or macro to run.

That is, [#DEFAULTS] invokes the keyword when the PMSEARCH command is executed, returning the current default volume and subvolume names and including them in the search list. #DEFAULTS is simply the keyword itself, which

PMSEARCH places in the search list intact; it is invoked later when an implied RUN command is executed, causing TACL to access the list.

The PMSEARCH command in the preceding example causes the program/macro search list to contain:

```
$SYSTEM.SYSTEM #DEFAULTS $OLD.HOME
```

At that time, #DEFAULTS also represents $OLD.HOME. The next VOLUME command changes the current default volume and subvolume; #DEFAULTS now represents $NEW.PLACE. The implied RUN command causes TACL to search $SYSTEM.SYSTEM, then $NEW.PLACE, and finally $OLD.HOME for the file EDIT.

- TACL invokes variables in preference to running programs. If, for example, you issue an implied RUN command using a name that identifies both a variable containing a TACL routine and a file containing an executable program, TACL executes the routine instead of the program.
- If you plan to run any system utilities that have duplicate names in other subvolumes in your list, place $SYSTEM.SYSTEM first in your search list.
- To see the contents of the program and macro search list, use the ENV Command on page 8-60 or the #PMSEARCHLIST Built-In Variable on page 9-285.

PMSG Command

Use the PMSG (process message) command to control logging of creation and deletion messages about processes you create with the RUN command.

```
PMSG { ON | OFF }
```

ON

specifies that whenever a process started by the current TACL process is created or deleted, a message is sent to the TACL OUT file (usually the home terminal).

OFF

specifies that process creation and deletion messages are not to be displayed. (OFF is the default setting.)

The form of process creation messages is:

PID: [*node-name*.]{$*process-name* | *cpu,pin* } *file-name*

node-name

is the name of the system where the process (represented by $*process-name* or by *cpu,pin*) is created.

$*process-name*

is the process name of the newly created process.

cpu,pin

is the CPU number and process number of the new process.

file-name

is the file name of the new process.

The forms of process deletion messages are:

STOPPED: [*node-name*.]{$*process-name* | *cpu,pin* }

ABENDED: [*node-name*.]{$*process-name* | *cpu,pin* }

node-name

is the name of the system in which the process stopped or abnormally ended.

$*process-name*

is the name of the process that is to be deleted.

cpu,pin

is the CPU number and process number of the process that is to be deleted.

Each process deletion message is followed by a CPU time-usage message on the next line. The form of CPU time-usage messages is:

```
CPU TIME: hh:mm:ss.fff
```

hh

is the number of hours since the process started.

mm

is the number of minutes past the number of hours since the process started.

ss

is the number of seconds past the number of hours and minutes since the process started.

ff

is the number of fractions of a second past the number of hours, minutes, and seconds since the process started.

Considerations

- TACL always displays an ABENDED message, even if PMSG is set to OFF.
- To determine the setting of PMSG, check the #PMSG built-in variable.

Example

This example shows the type of information you receive when you enter the PMSG ON command:

```
21> PMSG ON
22> EDIT
PID: 08,62
TEXT EDITOR - ...
*

 ***

*exit
STOPPED: 08,62
CPU TIME 0:00:00.019
```

The command at history number 21 turns on process messages. The command at number 22 starts an EDIT process; TACL displays the cpu,pin of the process as EDIT signs on. When you exit from the EDIT process, TACL displays a process-deletion message.

POP Command

Use the POP command to delete the top-level definition of a variable. The POP command is an alias for the #POP built-in function.

```
POP variable [ [,] variable ] ...
```

variable

is the name of an existing variable or the name of a built-in variable.

Considerations

- If the top level is the only level, the POP command deletes the variable, except for built-in variables. Trying to pop a variable that has not been pushed produces an “Expecting an existing variable” message. Trying to pop the last level of a built-in variable produces a “wasn’t pushed” message.
- When TACL encounters an #UNFRAME, it performs an implicit POP for every PUSH (or #PUSH) that was done since the most recent #FRAME was issued.
- TACL performs an implicit #POP #IN when #INPUT /UNTIL EOF/ or #INPUTV /UNTIL EOF/ is used.
- If TACL encounters an error, it performs an implicit #POP #OUT.
- Do not try to pop the root directory (:). If you try this, TACL returns:

  ```
  *ERROR* Cannot push or pop the root segment's root.
  ```

- In addition, to avoid losing standard functionality from your TACL environment, do not pop the directories supplied as part of a software RVU (such as UTILS).

POSTDUMP Utility

The POSTDUMP utility enables explicit postdump processing of an existing full dump file to produce an extracted dump file. It is provided for processing an existing full dump produced by either the TNet dump method or the PARALLEL dump method. You can use the FULL option of RCVDUMP to do a full dump, reload the halted processor, and then explicitly run the POSTDUMP utility to create an extracted dump so that the processor can be reloaded slightly sooner by skipping the implicit postdump processing step.

The POSTDUMP utility does not need to be licensed because it is allowed to run under any user ID.

```
POSTDUMP <in file> [ < options > ]
```

<in file>

is the name of an existing dump file used for input.

<options>

The format of *options* is:

```
OUT <out file>, [+]PIN { ALL | <pin list> },
             ALL_PAGES, [-]DP2_CACHE, [-]DP2_SHARED,
             [-]IMPLICIT_DLLS, [-]LOCKED, [-]OTHER,
              [-]SYS_LIBRARY, [-]SYS_PROCESS, [-]SYSTEM
```

`OUT <out file>`

specifies the name of the extracted dump file to be created. The default value is `<in file>` name suffixed with 'P', removing the third character, if necessary, to fit the resulting name within eight characters.

`[+]PIN { ALL | <pin list> }`

specifies the sponsor pin(s) for which all (except free) pages are to be included in the extracted dump. Default is processes related to the processor halt.

- + indicates that the specified pins are additional.
- ALL indicates all (valid) pins.
- <pin list> is a decimal number or is one or more decimal numbers, comma or space separated within parentheses.

`ALL_PAGES`

includes all (except free) pages for all (valid) pins.

`[-]DP2_CACHE`

Exclude | Include DP2 cache pages.

`[-]DP2_SHARED`

Exclude | Include DP2 shared segment pages.

`[-]IMPLICIT_DLLS`

Exclude | Include implicit DLL pages.

`[-]LOCKED`

Exclude | Include locked pages.

`[-]OTHER`

Exclude | Include possibly unclassified pages.

`[-]SYS_LIBRARY`

Exclude | Include system/public library pages.

`[-]SYS_PROCESS`

Exclude | Include sysgenned process pages.

`[-]SYSTEM`

Exclude | Include KSEG0/VKSEG64 pages.

Considerations

- HP recommends not specifying any of the POSTDUMP options other than OUT (to specifically name the output file) because the default page selection algorithm is the same as that used with the implicit postdump processing by RCVDUMP and is typically the best option.
- The POSTDUMP utility is provided to process an existing full dump produced by RCVDUMP using either the TNet dump method or the PARALLEL dump method. It will also accept as input an existing extracted dump or an existing full dump produced by the ONLINE dump method.

Examples

To process a full dump file CPU01 and create an extracted dump CPU01P, run the following command:

```
10> POSTDUMP CPU01
```

Note. Whether installed by default or explicitly installed, the POSTDUMP utility can be used to produce an extracted dump from a full dump of any NS-series processor that was running the H06.13 RVU or later, or from a full dump of any HP Integrity NonStop™ BladeSystems (which runs a J-series RVU).

The POSTDUMP utility can be run on such a system that was running the H06.13 RVU or later to process a full dump file from any other NS-series processor or a full dump file from any NonStop BladeSystems. The POSTDUMP utility does not rely on any of the other files (RCVDUMP, ONLINDMP, ZDMPDLL).

PPD Command

Use the PPD (process-pair directory) command to display the names, cpu,pin designations, and ancestors of one or more processes in the destination control table (DCT). The PPD command is an alias for the #XPPD built-in function.

```
PPD [ / OUT list-file / ]
  [ [ \node-name] [.{ $process-name | cpu,pin | * } ] ...]
```

`OUT list-file`

specifies a device, or a sequential file accessible to the sequential I/O (SIO) facility, that is to receive the output from PPD. If you omit this option, TACL writes the listing to its current OUT file.

If you specify an OUT file that does not exist, TACL creates an EDIT file named list-file. If you specify an OUT file that already exists, TACL appends the information to the end of the file.

`\node-name`

is the system where the process resides.

`$process-name`

is the name of the process or process pair.

`cpu,pin`

is the CPU number and process identification number of the process.

`*`

displays information about all processes, including I/O processes. (This applies to D-series or later systems only.)

Considerations

- If you do not specify a process (`$process-name` or `cpu,pin`), PPD lists all the current entries in the process-pair directory of the specified system. If you omit the node name, the default system is assumed.
- If you specify more than one process, PPD lists information for each process.
- The PPD command displays information as shown:

```
Name Primary Backup Ancestor
pname pcpu bcpu aid

pname pcpu bcpu aid
```

pname

is the name of the process. The name $Z000 identifies the initial process that was started when the system was cold loaded from disk.

pcpu

is the CPU number and process number of the primary process in a process pair, or of the specified process only if it is not a member of a pair.

bcpu

is the CPU number and process number of the backup process in a process pair. If backup is not displayed, the process has no backup.

aid

is the identity of the ancestor process; that is, the process that created the process listed under “Name.” If the ancestor is a named process, this field lists its name; otherwise, this field contains a CPU number and process number. If the ancestor process node is different from the specified or default node, then TACL displays the node name. The initial process that was started when the system was cold loaded from disk has no entry in this field.

There can be three states of an ancestor: AVAILABLE, UNAVAILABLE, and DEFUNCT.

- An ancestor process is called as DEFUNCT if it is present in the ancestor phandle. But currently it does not exist.
- When we run a process from one system to another, the ancestor of the process becomes UNAVAILABLE if they are disconnected manually.
- An ancestor is known to be AVAILABLE if it’s in a runnable condition.

If you include \ node-name in your PPD command, the name of the system is displayed at the beginning of the command listing.

- The DCT lists only named processes. To name a process that you start with a RUN command, include the NAME parameter in your command. (For more information about starting and naming processes, see the description of the RUN[D|V] Command on page 8-156.)
- The DCT lists named processes that do not have backups as well as named process pairs.

Examples

You can get a listing of all entries in the DCT and send it to the file PROCESS.PAIR by entering:

```
14> PPD / OUT PROCESS.PAIR /
```

To view the entry in the DCT for the process $WOW, enter:

```
15> PPD $WOW
Name Primary Backup Ancestor
$WOW 04,054 05,009 $Z000
```

PURGE Command

Use the PURGE command to delete one or more disk files.

```
PURGE [ / option / ]
  file-name-template [ , file-name-template ... ]
```

option

qualifies the purge operation. Specify option as one of these:

CONFIRM

specifies that TACL is to prompt for confirmation for each file, specified by name or by matching a template, before purging the file.

NOCONFIRM

specifies that TACL is to purge the file without requesting confirmation. This option is provided primarily for programmatic applications that do not need confirmation of the purge operation.

file-name-template

is a file name or partial file name with wild-card characters. For more information about file-name templates, see Section 2, Lexical Elements.

Considerations

These considerations apply to the PURGE command:

- When you use the PURGE command to delete a disk file, the entry for the file is deleted from the file directory in the volume where it resides, and any space previously allocated to that file is made available to other files. Data in the file is not physically deleted from the disk unless you specified the FUP CLEARONPURGE option when you created the file. Files that are physically deleted are overwritten with zeros. For information about the CLEARONPURGE option, see the *File Utility Program (FUP) Reference Manual*.
- You can purge a file only if it is not currently open; in addition, you must have purge access to the file. See the description of the DEFAULT Program on page 8-53 for information about file-access restrictions.
- If you try to use the PURGE command to delete a file that is being audited by the TMF subsystem the attempt fails if there are pending transaction-mode records or file locks; file-system error 12 (file in use) is returned. The purge is successful, however, if the audited file is inactive.

These considerations apply to the use of the CONFIRM and NOCONFIRM options:

- If you specify a file name with no wild-card characters and do not include CONFIRM or NOCONFIRM, TACL does not display a confirm prompt.

- If you specify a file-name template with wild-card characters and do not include CONFIRM or NOCONFIRM, TACL confirms only the template (not individual file names) before purging all files that match the template.
- CONFIRM and NOCONFIRM are mutually exclusive.

Examples

If you have purge access to the files OCT and NOV (in your current default system, volume, and subvolume) and to the file $RECORDS.DUE.PAST, you can purge all of them by entering:

```
14> PURGE OCT, NOV, $RECORDS.DUE.PAST
$DATA.LNP.OCT Purged
$DATA.LNP.NOV Purged
$RECORDS.DUE.PAST Purged
15>
```

This command requests confirmation of the PURGE operation:

```
15> PURGE /CONFIRM/ abc
PURGE $VOL1.SV.ABC (Y/[N])?
```

If you respond with Y, TACL purges the file. Any other response is treated as N.

This command confirms the file-name template before purging matching files:

```
16> PURGE abc*


PURGE $VOL1.SV.ABC* (y/[n])?
```

This command confirms each file that matches a file-name template before purging matching files:

```
17> PURGE /CONFIRM/ abc*
PURGE $VOL1.SV.ABC1 (y/[n])? y
$VOL1.SV.ABC2 Purged
PURGE $VOL1.SV.ABC2 (y/[n])? n
18>
```

This macro alters PURGE confirmation behavior so that TACL confirms purge operations:

```
?SECTION purgec MACRO
PURGE /CONFIRM/ %*%
```

This macro alters PURGE confirmation behavior so that TACL does not confirm purge operations:

```
?SECTION purgen MACRO
PURGE /NOCONFIRM/ %*%
```

To use either of the preceding macros, load the associated file and type PURGEC or PURGEN, respectively.

PUSH Command

Use the PUSH command to create a new top level for one or more variables or built-in variables. The PUSH command is an alias for the #PUSH built-in function.

```
PUSH variable [ [,] variable ] ...
```

variable

is a valid variable or a built-in variable name.

Considerations

- For existing variables that are not built-in variables, PUSH creates an empty top level of type TEXT.
- For built-in variables, PUSH creates a new top-level definition, copying the old top level to the new one (top-level and second-level definitions are now the same).
- If the variable does not exist, PUSH registers the name of the variable, but does not allocate space until you use SET VARIABLE or a similar command or built-in function to actually place data into the variable. As a result, you must perform a SET VARIABLE or related operation prior to using the variable in an #IF call or other command or function that tests the value of the variable.
- The default type for variables is TEXT. To change this type, use the SET VARIABLE command.
- Do not try to PUSH the root directory (:). If you try this, TACL returns "`*ERROR* Cannot push or pop the root segment's root.`" In addition, to avoid losing standard functionality from your TACL environment, do not PUSH the directories supplied as part of a software RVU (such as UTILS).

RCVDUMP Program (Super-Group or Super ID Only)

Use the RCVDUMP program to dump the memory of a processor to a disk file. The processor being dumped can be running or halted. The processor from which the RCVDUMP program is run must have X or Y fabric (G-series and H-series RVUs) or X bus or Y bus (D-series RVUs) access to the processor being dumped.

For G-series and H-series RVUs, you must have the super ID (255,255) to issue this command. For D-series RVUs, you must have a super-group user ID (255,*user-id-number*).

Beginning with the H06.07 RVU in H-series releases and with all J-series RVUs, RCVDUMP and ONLINDMP (which is run by RCVDUMP to do an ONLINE dump or a PARALLEL dump) terminate immediately, if run under other than a super-group user ID. As normally installed, RCVDUMP can only be run under the super ID (255,255). If on a particular system you want to allow RCVDUMP to be run under any super-group user ID (255, *user-id-number*), you can use FUP to license the RCVDUMP and ONLINDMP files located in the relevant $SYSTEM.SYSnn subvolume.

When a processor is dumped to disk, the RCVDUMP utility copies the dump in a compressed format from the specified processor to a disk file.

Pressing the BREAK key while the RCVDUMP program is running causes RCVDUMP to abend or stop, and leaves the memory dump unfinished. If this happens, you must purge the dump file and restart the RCVDUMP program

The hardware architecture of H-series systems requires a change of terminology. H-series uses logical processors that consist of one to three physical processors known as processor elements (PEs). The logical processor is what used to be called the CPU. For availability issues, the processor elements are all located on different circuit boards. These boards are known as blades (or slices) and are identified by the letters A, B, or C. With this particular command you may be concerned with dumping one or more PEs from one or more slices. The *PARALLEL* method is a time-saving way to dump the memory of a single PE.

Extracted Memory Dump

The RCVDUMP utility enables dumping of a subset of the full memory in an extracted dump file that should almost always be sufficient to analyze the processor halt. An extracted dump file may or may not be created in addition to or instead of a full dump file. An extracted dump file can be transferred to the Global Mission Critical Solution Center (GMCSC) or Development in a shorter time than it would take to transfer a full dump file. When a full dump file is not created, overall dumping elapsed time and overall disk space used are also reduced. If it is determined that a useful extracted dump file cannot be created, no error is reported and only a full dump file is created instead.

The RCVDUMP HELP output summarizes when a full dump file or an extracted dump file or both are created.

If on a non-NSAA system, the halted processor to be dumped is running at minimum the version of Halted State Services included in either the H06.17 RVU on H-series or the J06.06 RVU on J-series, RCVDUMP by default creates an extracted dump file only, without first creating a full dump file.

If a full memory dump is to be created, it is created first, followed by implicit postdump processing of the full dump file to produce a smaller extracted dump file. Although this additional processing step increases the overall dumping elapsed time and increases the overall disk space used for the dump files, the benefit is in transferring only the smaller extracted dump file to GMCSC. The full dump file remains available in case it needs to be transferred later if information not included in the extracted dump turns out to be required for the halt analysis.

After performing a TNet dump (full dump), the implicit postdump processing is executed, by default, when RCVDUMP runs either on an NSAA or non-NSAA system. On an NSAA system, implicit postdump processing is executed, by default, following a PARALLEL dump. Implicit postdump processing is not executed following an ONLINE dump, because the full dump may not provide a complete and consistent set of information required to select a subset for an extracted dump.

RCVDUMP supports a new option, FULL, that suppresses creation of an extracted dump file and results in creation of only a full dump file. If only an extracted dump file is created and the extracted dump information turns out to be insufficient to analyze the processor halt, it may be necessary to reproduce the halt and use the FULL option of RCVDUMP to create a full dump file to be used for the analysis.

Note. When both full and extracted dump files are created, the full dump file name is changed to end with an 'F'.

The files (RCVDUMP, ONLINDMP, POSTDUMP, ZDMPDLL) related to the NonStop operating system Utilities product, from the H06.14 RVU, can be also be installed to provide implicit and explicit postdump processing functionality on an HP Integrity NonStop (NS-series) system running the H06.13 RVU, which does not already have this functionality by default.

Neither implicit nor explicit postdump processing is supported for a dump of an NS-series processor that was running a release earlier than the H06.13 RVU, because insufficient information is available in the full dump to appropriately select a subset for an extracted dump. For this reason, it is not useful to install the new files on a system running any release earlier than the H06.13 RVU.

△ **Caution.** This command should be used only as part of a documented processor dumping and recovery procedure. See the sections Processors: Monitoring and Recovery and Starting and Stopping the System in the *NonStop S-Series Operations Guide* or the *Integrity NonStop NS-Series Operations Guide*.

If a memory dump is requested as part of a recovery procedure, for processors with more than 2 gigabytes of memory, use the HP Tandem Failure Data System (TFDS) memory dump facility to ensure that the dump occurs in a timely fashion. Otherwise, use either RCVDUMP or the TFDS facility. TFDS must be configured on the system before the halt occurs.

H-Series Syntax

```
RCVDUMP   <filename>, cpuNum [, SLICE <sliceId>]
[, START <startAddress>][, END <endAddress>]
[, [ONLINE | PARALLEL] [, FULL] ]
```

filename

is the name of the disk file to which the dump is to be written.

cpuNum

is the number of the logical processor from which a processor element is to be dumped. Specify *cpuNum* as an integer in the range from 0 through 15.

SLICE <sliceId>

is the identification of the slice from which the processor element is to be dumped. Valid values are *A* or *B* or *C* or *ALL*. The default is *ALL*. Note that *ALL* may not be used with the parallel method of dumping.

START n...

is the byte address where the dump will start. The default value is 0.

END n...

is the byte address where the dump will stop. Using a value of -1 is the same as specifying the end of memory. The default value is -1.

ONLINE

If this option is specified, the dump of a processor can be taken while it is running and only a full dump file (excluding free memory) is created. You may use either *PARALLEL* or *ONLINE* but not both.

PARALLEL

If this option is specified, the dump of a single processor element can be taken while the other PEs in that logical processor are reloaded and continue normal

operations. By default, both the full dump and extracted dump files are created. You may use either *PARALLEL* or *ONLINE* but not both.

Note. When neither *ONLINE* nor *PARALLEL* is specified:

- On an NSAA system, a full dump file is always created. An extracted dump file is created, by default.
- On a non-NSAA system, a full dump file is not created if the processor to be dumped is running a version of Halted State Services that supports direct creation of an extracted dump without a full dump. Otherwise, a full dump file is created. An extracted dump file is created, by default.

FULL

Specifies that only a full dump is to be created and not an extracted dump.

Note. This option is supported only on systems running J06.03 and later J-series RVUs and H06.14 and later H-series RVUs.

G-Series Syntax

```
RCVDUMP [ / run-option [ , run-option ] ... / ]
  dump-file , cpu , [ fabric, param [ , param ] ]
```

run-option

is any of the options described in the RUN[D|V] Command on page 8-156.

dump-file

is the name of the disk file to which the dump is to be written.

cpu

is the number of the processor that is to be dumped. Specify *cpu* as an integer in the range from 0 through 15.

fabric

specifies the ServerNet fabric to be used for the dump operation. The default fabric is the X fabric. Possible values are:

X = X fabric

Y = Y fabric

param

is one or more of these parameters:

START *n...*

is the byte address where the dump will start. The default value is 0.

END *n...*

is the byte address where the dump will stop. Using a value of -1 is the same as specifying the end of memory. The default value is -1.

ONLINE

If this option is specified, a dump can be taken of a processor while it is running. Only memory pages are included in this type of dump. No registers, translation lookaside buffers (TLB) or caches are dumped. All other parameters are ignored, including *fabric*.

RESET

If this option is specified, a soft reset is issued to the processor being dumped. This option is necessary if the processor has experienced a hardware error freeze instead of a halt.

Note. The PRIME and NOPRIME parameters do not have any effect on how the RCVDUMP command operates. They are therefore no longer described in the RCVDUMP command description.

D-Series Syntax

```
RCVDUMP [ / run-option [ , run-option ] ... / ]
  dump-file , cpu , [ bus, param [ , param ] ]
```

run-option

is any of the options described in the RUN[D|V] Command on page 8-156.

dump-file

is the name of the disk file to which the dump is to be written.

cpu

is the number of the processor that is to be dumped. Specify *cpu* as an integer in the range from 0 to 15, inclusive. This processor must be primed for a memory bus dump before you attempt the dump.

bus

specifies the bus to be used for the dump. The default bus is the X bus. Possible values are:

0 = X bus

1 = Y bus

param

is either one of these parameters:

FULL

specifies that the entire physical memory is to be dumped. FULL is the default for a processor if the size of physical memory is less than or equal to sixteen megabytes. This option is ignored if the processor is not a VLX.

PARTIAL

specifies that only those pages of physical memory that are mapped in the page table cache are to be dumped. PARTIAL is the default for a processor with more than sixteen megabytes of physical memory. This option is ignored if the processor is not a VLX.

Note. The PRIME and NOPRIME parameters do not have any effect on how the RCVDUMP command operates. They are therefore no longer described in the RCVDUMP command description.

Considerations

- If the file *dump-file* does not already exist, a file with that name is created.

  D-series: If *dump-file* exists, it must be empty (its end-of-file pointer must be set to zero). If *dump-file* is not empty, RCVDUMP aborts. Also, if the empty *dump-file* exists, but its primary and secondary extent sizes are too small to contain the entire dump, the file is purged, and a new *dump-file* with extent sizes of sufficient size is created.

  H-series or G-series: If the file already exists RCVDUMP prompts the user to specify whether it can overwrite the file. If the answer is ' no', RCVDUMP aborts. Otherwise it overwrites the existing file and continues.

- Any dump file created with G06.16 RVU or later has a file code of 145. Any dump file created with an earlier RVU has a file code of 144. Any dump file created with any J-series RVU or any H-series RVU has a file code of 148.

- In H-series, if the system is not a duplex or triplex configuration, or if the *ONLINE* or *PARALLEL* method is not specified, the default method is to use the TNet dump method in conjunction with Halted State Services running in the halted processor to be dumped. For more information on Halted State Services, contact your local HP service provider.

Example

For G-series RVUs, if you have the super-group user ID, you can initiate a dump from processor 6 of your system over the X fabric and send the dump to file $SYSTEM.DUMP.DUMP1 by entering:

```
14> RCVDUMP $SYSTEM.DUMP.DUMP1 , 6 , X
CPU 06 HAS BEEN DUMPED TO $SYSTEM.DUMP.DUMP1.
```

For H-series RVUs, if you have the super-group user ID, you can initiate a dump from the processor element on slice A that belongs to logical processor 3, while the other processor elements of logical processor 3 continue normal operations:

```
10> RCVDUMP $SYSTEM.DUMP.DUMP1 , 3 , SLICE A , PARALLEL
```

RECEIVEDUMP Command (Super-Group Only)

Use the RECEIVEDUMP command to dump the memory of a processor to a disk file. The processor being dumped can be running or halted. This command executes the RCVDUMP program. The processor from which the RECEIVEDUMP command is run must have X or Y fabric (G-series RVUs) or X bus or Y bus (D-series RVUs) access to the processor being dumped.

This command is not supported in H-series systems, use RCVDUMP instead.

For G-series RVUs, you must have the super ID (255,255) to issue this command. For D-series RVUs, you must have a super-group user ID (255,*user-id-number*).

When a processor is dumped to disk, the RCVDUMP utility copies the dump in a compressed format from the specified processor to a disk file.

Pressing the BREAK key while the RECEIVEDUMP command is running causes RCVDUMP to abend or stop, and leaves the memory dump unfinished. If this happens, you must purge the dump file and restart the RECEIVEDUMP command.

△ **Caution.** This command should be used only as part of a documented processor dumping and recovery procedure. Refer to the sections Processors: Monitoring and Recovery and Starting and Stopping the System in the *NonStop S-Series Operations Guide*.

If a memory dump is requested as part of a recovery procedure, for processors with more than 2 gigabytes of memory, use the TFDS memory dump facility to ensure that the dump occurs in a timely fashion. Otherwise, use either RCVDUMP or the TFDS facility. TFDS must be configured on the system before the halt occurs.

G-Series Syntax

```
RECEIVEDUMP / OUT dump-file / cpu , fabric
  [ , param [ , param ] ]
```

dump-file

is the name of the disk file to which the dump is to be written.

cpu

is the number of the processor that is to be dumped. Specify *cpu* as an integer in the range from 0 through 15.

fabric

specifies the ServerNet fabric to be used for the dump operation. The default fabric is the X fabric. Possible values are:

0 = X fabric

1 = Y fabric

param

is one or both of these parameters:

ONLINE

If this option is specified, a dump can be taken of a processor while it is running. Only memory pages are included in this type of dump. No registers, translation lookaside buffers (TLB) or caches are dumped. All other parameters are ignored, including *fabric*.

RESET

If this option is specified, a soft reset is issued to the processor being dumped. This option is necessary if the processor has experienced a hardware error freeze instead of a halt.

Note. The PRIME and NOPRIME parameters do not have any effect on how the RCVDUMP command operates. They are therefore no longer described in the RCVDUMP command description.

D-Series Syntax

```
RECEIVEDUMP / OUT dump-file / cpu , bus
  [ , param [ , param ] ]
```

dump-file

is the name of the disk file to which the dump is to be written.

cpu

is the number of the processor that is to be dumped. Specify *cpu* as an integer in the range from 0 to 15, inclusive. This processor must be primed for a memory bus dump before you attempt the dump.

bus

specifies the bus to be used for the dump. The default bus is the X bus. Possible values are:

0 = X bus

1 = Y bus

param

is either one of these parameters:

FULL

specifies that the entire physical memory is to be dumped. FULL is the default for a processor if the size of physical memory is less than or equal to sixteen megabytes. This option is ignored if the processor is not a VLX.

PARTIAL

specifies that only those pages of physical memory that are mapped in the page table cache are to be dumped. PARTIAL is the default for a processor with more than sixteen megabytes of physical memory. This option is ignored if the processor is not a VLX.

Note. The PRIME and NOPRIME parameters do not have any effect on how the RCVDUMP command operates. They are therefore no longer described in the RCVDUMP command description.

Considerations

- If the file *dump-file* does not exist when you enter the RECEIVEDUMP command, a new file named *dump-file* is created.

  D-series: If *dump-file* exists, it must be empty; that is, its end-of-file (EOF) pointer must be set to zero. If *dump-file* is not empty, it is not overwritten: RECEIVEDUMP returns an error message. Also, if the empty *dump-file* exists, but its primary and secondary extent sizes are too small to contain the entire dump, the file is purged and a new *dump-file* with extent sizes of sufficient size is created.

  G-Series: If the file already exists RCVDUMP prompts the user to specify whether it can overwrite the file. If the answer is ' no', RCVDUMP aborts. Otherwise it overwrites the existing file and continues.

- Any dump file created with RVU G06.16 or later has a file code of 145. Any dump file created with an earlier RVU has a file code of 144.

Example

For G-series RVUs, if you have a super-group user ID, you can initiate a dump from processor 4 of your system over the Y fabric and send the dump to file $SYSTEM.DUMP.DUMP2 by entering:

```
74> RECEIVEDUMP /OUT $SYSTEM.DUMP.DUMP2/ 4,1
CPU 04 HAS BEEN DUMPED TO $SYSTEM.DUMP.DUMP2.
```

RELOAD Program (Super-Group Only)

Run the RELOAD program to reload the remaining processors after the first processor in a system has been brought up, or to recover a processor that has failed. You must use a super-group user ID (255,*your-id*) to issue this command.

This command should be used only as part of a documented system startup or processor recovery procedure. Refer to the sections Processors: Monitoring and Recovery and Starting and Stopping the System in the *NonStop NS-Series Operations Guide* or *NonStop S-Series Operations Guide* for further details.

The hardware architecture of H-series systems requires a change of terminology. There are now logical processors that consist of one to three physical processors known as processor elements (PEs). The logical processor is what used to be called the CPU. For availability issues, the processor elements are all located on different circuit boards. These boards are known as blades (or slices) and are identified by the letters A, B, or C.

The H-series RELOAD program allows you to omit a selected slice from the reload. This will prevent the selected PE on that slice from reloading. Other PEs on that slice belong to other logical processors and therefore remain unaffected by these operations.

You may then perform a parallel dump of that PEs memory while continuing to perform normal operations on the other newly-reloaded PEs within that logical processor. The parallel method is a time-saving way to dump the memory of a single PE. See the RCVDUMP Program (Super-Group or Super ID Only) on page 8-132 for further details.

Note that in the following H-series syntax all references to processor are references to the logical processor or CPU, except where the term processor element or PE is explicitly used.

H-Series Syntax

```
RELOAD [ / run-option [ , run-option ] ... / ]
  cpu-set [; cpu-set ] ...
```

run-option

is any of the options described in the RUN[D|V] Command on page 8-156.

cpu-set

is a set of processors (and associated options) to be reloaded. Specify *cpu-set* as:

```
{ cpu-range } [, option, option, ... ]
{ ( cpu-range, cpu-range, ...) }
{ * }
```

cpu-range

is one of these:

```
cpu
cpu-cpu
```

cpu

is the processor number, an integer from 0 through 15.

cpu-cpu

is two processor numbers separated by a hyphen, specifying a range of processors. In a range specification, the first processor number must be less than the second.

option

is one of these:

```
NOSWITCH
[PRIME|NOPRIME]
<fabric>
OMITSLICE [A|B|C]
<$volume [.sysnn.osdir]>
```

NOSWITCH

specifies that, when a processor is reloaded, there is no default autoswitch of controller ownership to the configured primary processor.

[PRIME|NOPRIME]

Sets up the logical processor for the reload operation.

NOPRIME is the default.

fabric

specifies whether the X fabric or Y fabric is used for the transfer of the operating system image to the processor during the RELOAD operation.

0 = X fabric
1 = Y fabric

The default option is the X fabric.

`OMITSLICE [A|B|C]`

The PE on the selected slice will not be reloaded when other PEs in that logical processor are reloaded. If you do not provide an argument (A or B or C) the system will choose a slice to omit.

`<$volume [.sysnn.osdir]>`

specifies a volume other than $SYSTEM where the operating system image (`sysnn.osdir`) to be used for reloading the processor is located.

This specification could take the form of `$volume` or `$volume.sysnn.osdir` or `sysnn.osdir` or `osdir` depending on your requirements.

*

specifies that all failed processors should be reloaded.

A help screen is displayed if you enter RELOAD with no parameters and no IN file-name.

G-Series Syntax

```
RELOAD [ / run-option [ , run-option ] ... / ]
  cpu-set [; cpu-set ] ...
```

run-option

is any of the options described in the RUN[D|V] Command on page 8-156.

cpu-set

is a set of processors (and associated options) to be reloaded. Specify *cpu-set* as:

```
{ cpu-range } [, option, option, ... ]
{ ( cpu-range, cpu-range, ...) }
{ * }
```

cpu-range

is one of these:

```
cpu
cpu-cpu
```

cpu

is the processor number, an integer from 0 through 15.

cpu-cpu

is two processor numbers separated by a hyphen, specifying a range of processors. In a range specification, the first processor number must be less than the second.

`option`

is one of these:

```
NOSWITCH
[PRIME|NOPRIME]
<fabric>
<$volume [.sysnn.osimage]>
```

`NOSWITCH`

specifies that, when a processor is reloaded, there is no default autoswitch of controller ownership to the configured primary processor.

`[PRIME|NOPRIME]`

Sets up the logical processor for the reload operation.

NOPRIME is the default.

fabric

specifies whether the X fabric or Y fabric is used for the transfer of the operating system image to the processor during the RELOAD operation.

0 = X fabric
1 = Y fabric

The default option is the X fabric.

`<$volume [.sysnn.osimage]>`

specifies a volume other than $SYSTEM where the operating system image (SYS*nn*.OSIMAGE) to be used for reloading the processor is located.

This specification could take the form of `$volume` or `$volume.sysnn.osimage` or `sysnn.osimage` or `osimage` depending on your requirements.

If a subvolume is not specified, the default subvolume in effect at the time the RELOAD command is issued is used.

*

specifies that all failed processors should be reloaded.

A help screen is displayed if you enter RELOAD with no parameters and no IN file-name.

D-Series Syntax

```
RELOAD [ / run-option [ , run-option ] ... / ]
  cpu-set [; cpu-set ] ...
  [ HELP ]
```

run-option

is any of the options described in the RUN[D|V] Command on page 8-156.

cpu-set

is a set of processors (and associated options) to be reloaded. Specify *cpu-set* as:

```
{ cpu-range } [, option, option, ... ]
{ ( cpu-range, cpu-range, ...) }
{ * }
```

cpu-range

is one of these:

```
cpu
cpu-cpu
```

cpu

the processor number, an integer from 0 through 15.

cpu-cpu

two processor numbers separated by a hyphen, specifying a range of processors. In a range specification, the first processor number must be less than the second.

option

is one of these:

```
NOSWITCH
bus
$volume
subvolume.filename
```

NOSWITCH

specifies that, when a VLX, CLX, CYCLONE or NSR-L processor is reloaded, there is no default autoswitch of controller ownership to the configured primary processor.

bus

specifies whether the X bus or the Y bus is used for the transfer of the operating-system image to the processor during the RELOAD operation.

0 = X bus
1 = Y bus

The default option is the X bus.

$*volume*

specifies a volume other than $SYSTEM where the operating system image (SYS*nn*.OSIMAGE) to be used for reloading the processor is located.

subvolume.filename

specifies a subvolume and file where the operating system image to be used for reloading the processor is located. If *subvolume* is not specified, the default subvolume in effect at the time the RELOAD command is issued is used.

*

specifies that all failed processors should be reloaded.

HELP

displays a help screen summarizing the RELOAD command syntax and giving an example of RELOAD. The help screen is also displayed if you enter RELOAD with no parameters and no IN file-name.

Note. The PRIME and NOPRIME parameters do not have any effect on how the RCVDUMP command operates. They are, therefore, no longer described in the RCVDUMP command description.

Considerations

- To use the RELOAD command, you must have a group ID of 255.

- If your command repeats a processor number, RELOAD displays:

  ```
  CPU n already specified
  ```

  where *n* is the processor number you specified more than once in the RELOAD command.

- The alternate operating-system image file (in H-series, the OS FileSet) option:
  - Provides additional fault tolerance if the current file is unavailable.
  - Distributes paging activity.

  The alternate operating-system image file (in H-series, the OS FileSet) you use must be an exact duplicate of the file being used by the processor from which the RELOAD command is issued. All processors must be loaded with identical operating-system images. The alternate operating-system image file (in H-series, the OS FileSet) option is not a substitute for cold loading the system. It cannot be used to change the operating-system image.

  If an alternate operating-system image file (in H-series, the OS FileSet) is specified, that file continues to be used by the processor into which it is loaded. Thus, different processors can be using identical files from various locations.

  In addition, if an alternate operating-system image file (in H-series, the OS FileSet) is specified, all the files in the reloader's default SYSnn must be present in the subvolume specified in the alternate fileset option of the RELOAD command.

- G-series and D-series Only. RELOAD checks the operating-system image file to ensure that it has the same timestamp as the operating-system image in the processor doing the reload. If the timestamps do not match, and the volume is already accessible (allowing the timestamp to be immediately checked), reloading of the affected processors does not occur. If the volume is inaccessible, the reloaded processor checks the timestamp and halts with a %4016 code if it does not match.

- You can use an IN file to enter RELOAD command parameters. The IN file can be a disk file but not a terminal. An IN file can contain multiple lines as long as no `cpu-set` is divided between lines. To start a RELOAD process using an IN file, enter RELOAD with only the IN file-name specification. (For more information, see the RUN[D|V] Command on page 8-156.)

- When one of the two processors connected to a controller fails, the other processor gains ownership of the controller. When a failed processor (or one that has not been loaded) is reloaded:
  - If you specify NOSWITCH, no controller ownership switch takes place. The processor that did not fail retains ownership.
  - If you omit NOSWITCH, controller ownership action depends on the type of controller and whether the processor that was reloaded was configured during system-image generation as the primary processor for the controller. For controllers connected to terminals, serial printers, and data communication

lines, no controller ownership switch takes place. For these controllers, problems can arise if an automatic ownership switch occurs during data transmission. For other types of controllers (such as those for disks, tape drives, card readers, and nonserial printers), a controller ownership switch occurs if the reloaded processor was configured as the primary processor for the controller.

△ **Caution.** For NonStop VLX systems, when using RELOAD in the CIIN file, you should not use the PRIME option for two reasons: first, the processors are primed automatically in a system cold load. Second, if you are reloading a single processor, the PRIME option in the CIIN file causes any other failed processors to come up, thereby erasing their memory contents; you are then unable to dump these processors to analyze the failure.

- The procedures for dumping an entire system that is frozen are described in the *NonStop S-Series Operations Guide* or the *NonStop NS-Series Operations Guide*.

Note. When using the CIIN file, you should include in it only those necessary commands that are restricted to super-group use, such as SETTIME or ADDDSTTRANSITION, and perhaps a command to start a TACL process. Any other commands can be put into system startup files. In particular, you should not put commands to start application processes in the CIIN file, as they can cause CPU halts or system freezes in the event of application malfunction.

Examples

1. To reload processor 1 using the currently selected bus and to prevent switches in device ownership, enter:

```
14> RELOAD 1, NOSWITCH
```

2. This G-series example uses the Y bus and the alternate operating system image in the file $DATA.SYS02.OSIMAGE to reload processors 2, 4, 5, 6, and 7. The same command then reloads all other downed CPUs using the default operating system image and the currently selected bus:

```
15> RELOAD (2, 4-7) , $DATA.SYS02.OSIMAGE , 1; *
```

3. These H-series examples demonstrate use of the `OMITSLICE` option.

   The first command uses a mutiple *`cpu-set`* specification, reloading all processor elements belonging to logical processor 2 with the exception of the PE on slice A, and the command continues to a second *`cpu-set`* specification to reload all PEs belonging to processor 3.

```
2> reload 2, omitslice A; 3

NONSTOP OS  PROCESSOR RELOAD - T9070H02 - (01MAY05)

Reload 2: Fabric: 0 (X)

Reload 3: Fabric: 0 (X)

Omitted slice. (CPU 2, Slice A)
Sent reload start-up packet to cpu 2
Sent reload start-up packet to cpu 3
Sending OS FileSet pages to Reloadee(s)

Integrating CPU 2
Starting system services on CPU 2
PROCESSOR RELOAD: 2

Services started successfully.

Integrating CPU 3
Starting system services on CPU 3
PROCESSOR RELOAD: 3

Services started successfully.

CPU 2: reloaded.
CPU 3: reloaded.
```

The second example shows a reload of processor 3, with an `omitslice` option that allows the system to determine which slice to omit.

```
5> reload 3, omitslice

NONSTOP OS  PROCESSOR RELOAD - T9070H02 - (01MAY05)

Reload 3: Fabric: 0 (X)

Omitted slice. (CPU 3, Slice B)
Sent reload start-up packet to cpu 3
Sending OS FileSet pages to Reloadee(s)

Integrating CPU 3
Starting system services on CPU 3
PROCESSOR RELOAD: 3

Services started successfully.
CPU 3: reloaded.
```

REMOTEPASSWORD Command and RPASSWRD Program

Use the REMOTEPASSWORD command to run the system program RPASSWRD, which adds or deletes remote passwords.

```
REMOTEPASSWORD [ \node-name [ , password ] ]
```

\node-name

is the name of the system where the remote password is to be in effect. If this is your local system, then remote users with your user ID (normally, you) can access your local system if they have set the same remote password for*\node-name* on the remote system.

Conversely, if \ node-name is the name of a remote system, you can gain access to that system if password matches the remote password established for *\node-name* by a person with your user ID on that system.

password

is the remote password. It can contain from one to eight alphanumeric, nonblank characters. The password is case-sensitive; lowercase letters are not changed to uppercase.

Considerations

- If you omit password, your remote password for *\node-name* is deleted. If you omit *\node-name*, all of your remote passwords are deleted

  If you omit both parameters, TACL returns this prompt:

  ```
  Do you really want to delete all of your remote passwords?
  ```

  If you type y or yes (either uppercase. lowercase, or any combination of uppercase and lowercase characters), password deletion occurs.

  RPASSWRD also prompts first to verify that deleting your remote passwords is what you actually intend to do.

- You must establish remote passwords before you can access remote systems in a network. To gain access to a remote system, you must have identical remote passwords in effect on both the system where you are logged on and the remote system.
- If RPASSWRD detects a null character in a remote password, it aborts without changing the remote password. Null characters can inadvertently be included by entering certain control-character combinations at a terminal. See the user documentation for your terminal for a list of control characters that might interfere with your terminal operation.
- See the *Guardian User's Guide* for information about the use of remote passwords in network security.

Examples

1. Suppose that you are user MANUF.FRED on system \ROME, and you need access to files on system \PARIS. First, log on to the \ROME system and establish the remote passwords YES for the \PARIS system and OK for the \ROME system by entering these commands:

   ```
   14> REMOTEPASSWORD \PARIS, YES
   THE \PARIS REMOTE PASSWORD FOR MANUF.FRED (8,44) HAS BEEN
   CHANGED.
   15> REMOTEPASSWORD \ROME, OK
   THE \ROME REMOTE PASSWORD FOR MANUF.FRED (8,44) HAS BEEN
   CHANGED.
   16>
   ```

   Now you must log on to the \PARIS system and use the same commands to establish the same remote passwords. Because you do not yet have access to \PARIS, you normally need to contact the system manager for the \PARIS system, who logs on as a super-group user. The system manager then logs on with your user name and enters the commands for you.

2. To delete the remote password for \PARIS from the \ROME system, log on to \ROME and enter:

   ```
   20> REMOTEPASSWORD \PARIS
   THE \PARIS REMOTE PASSWORD FOR MANUF.FRED (008,044) HAS
   BEEN DELETED.
   21>
   ```

RENAME Command

Use the RENAME command to change the name of an existing disk file.

```
RENAME old-file-name [,] new-file-name
```

old-file-name

is the name of the disk file to be renamed.

new-file-name

is the new name for the file.

Considerations

- You can rename a file only if it is not open with exclusive access, and you either have purge access to the file or are logged on as a super-group user.
- You can use the RENAME command to change the subvolume name of a file, but not its volume name. Disk files that are renamed stay on the same disk volume.To change the volume where a file resides, copy the file to a new volume with the FUP DUP command, then delete the original file. For details, see the *File Utility Program (FUP) Reference Manual*.
- If you try to rename a file being audited by TMF, the attempt fails and file-system error 80 (operation invalid) is returned.For information about TMF and the AUDIT option, see the *TMF Reference Manual*.
- Format 1 files, Format 2 files, and files opened with READ ONLY ACCESS can be renamed.

Examples

1. To rename the file RECORDS.DATA to STORAGE.OLDDATA, enter:

```
14> RENAME RECORDS.DATA, STORAGE.OLDDAT
15>
```

2. To rename the file MYSTUFF in the current default subvolume to be TRASH in the subvolume EXTRA, enter:

```
15> RENAME MYSTUFF, EXTRA.TRASH
16>
```

RESET DEFINE Command

Use the RESET DEFINE command to restore one or more DEFINE attributes in the working attribute set to their initial settings. For more information about DEFINEs, see Section 5, Statements and Programs and the ADD DEFINE Command on page 8-9.

```
RESET DEFINE { attribute-name [, attribute-name ] ... }
               { * }
```

attribute-name

is the name of a DEFINE attribute whose value is to be reset to its initial value. For the syntax of attribute-name, see SET DEFINE Command on page 8-173. If you reset a defaulted attribute, it assumes its default value. If you reset an optional attribute, it has no value. You cannot reset a required attribute after a value has been assigned to it.

*

resets all the attributes in the working attribute set to their initial settings. That is, CLASS is reset to MAP, and the only CLASS MAP attribute, FILE, is reset to have no value.

Considerations

- If any error occurs on a RESET DEFINE command, the command has no effect on the working attribute set. The DEFINE command error messages are listed and described in Appendix B, Error Messages. Entering RESET DEFINE * is equivalent to specifying SET DEFINE CLASS MAP.
- That is, RESET DEFINE * resets CLASS to MAP (the default value), establishes the attributes of CLASS MAP as the working attribute set, and restores each of those attributes to its initial setting; for example:

```
69> RESET DEFINE *
70> SHOW DEFINE
    CLASS MAP
    FILE ??
Current attribute set is incomplete
```

- Because the CLASS attribute acts as a DEFINE subtype, you should keep these points in mind when using the RESET DEFINE command:
  - Attributes are reset in the order they are specified in your command.
  - Resetting the CLASS attribute establishes a new working attribute set of the default CLASS (MAP); its single attribute, FILE, has no value.
  - An error occurs if you reset an attribute that is not a member of the working attribute set (such as an attribute that is not associated with the current CLASS).

Example

This RESET DEFINE command resets the USE attribute in the current working attribute set. Because USE is an optional attribute, it ceases to have any value.

```
71> SHOW DEFINE *
    CLASS                TAPE
    VOLUME               M5436
    LABELS               IBM
    REELS
    OWNER                PURCHG
    FILESECT             002
    ...
    USE                  IN
    DEVICE               $TAPE
    ...
    TAPEMODE
72> RESET DEFINE USE
73> SHOW DEFINE *
    CLASS                TAPE
    VOLUME               M5436
    LABELS               IBM
    REELS
    OWNER                PURCHG
    FILESECT             002
    ...
    USE
    DEVICE               $TAPE
    ...
    TAPEMODE
```

RUN[D|V] Command

Use the RUN command to run programs or TACL macros. You invoke the RUND or RUNV command with the same parameters and options as the RUN command, but the RUND command runs programs under control of the INSPECT symbolic debugger or DEBUG, while the RUNV command runs programs under the control of the VISUAL INSPECT symbolic debugger.

A RUN command must name an object file or TACL program file that contains the program you want to run. You can enter RUN commands either explicitly or implicitly:

For an explicit RUN command, enter the keyword RUN (or RUND or RUNV) followed by the name of the program file.

For an implicit RUN command, enter the name of the program file. If you omit RUN or enter a command that TACL does not recognize, TACL assumes you are entering an implicit run command. TACL searches for program-file in the subvolumes listed in the #PMSEARCHLIST variable

The program DEBUG is not available for use on systems running H-series software.

The DEBUG command invokes a debugger, it can be Inspect, Native Inspect (eInspect, which is not in the family of Inspect debuggers), or Visual Inspect.

If the INSPECT attribute is set ON anywhere (in the object file during compilation, or on the TACL command line using the SET command), you will get a debugger in the Inspect family (either Inspect or VI), unless of course neither of these debuggers is available, and then you get the default debugger, eInspect. If the Inspect attribute is OFF, you get Native Inspect (eInspect).

Inspect is invoked only for TNS accelerated/interpreted programs (never for TNS/E native programs), while Visual Inspect can handle both of these. Native Inspect handles only TNS/E native programs and snapshots..

```
[ RUN[D] ] program-file
[/ run-option [ , run-option ] ... / ]
[ param-set ]
```

program-file

is the name of the file containing the object program to be run. Partial file names are expanded using the current TACL default system, volume, and subvolume names if you enter an explicit RUN command.

run-option

is any of these:

CPU	HIGHPIN	JOBID	NAME	PRI
DEBUG	IN	LIB	NOWAIT	STATUS
DEFMODE	INLINE	MAXMAINSTACKSIZE*	OUT	SWAP

EXTSWAP	INSPECT	MAXNATIVEHEAPSIZE*	OUTV	TERM
GUARANTEED-SWAPSPACE*	INV	MEM	PFS	WINDOW

* These options are available when the default configuration settings of the TACL process are changed. To change the settings, set the TACL configuration parameter CONFIGRUN to PROCESSLAUNCH. For more information, see the #SETCONFIGURATION option, CONFIGRUN [PROCESSCREATE | PROCESSLAUNCH].

`CPU` *`cpu-number`*

specifies the number of the processor where the process is to run. Specify *`cpu-number`* as an integer in the range from 0 through 15. If you omit this option, the process runs in the same processor as TACL (but if a $CMON process exists, it might specify a processor other than that one; see the *Guardian Programmer's Guide* for information about $CMON).

`DEBUG`

causes the process to start in the debug mode. This option is not valid in a RUND or RUNV command.

RUN *`program-file`* / DEBUG / is the same as RUND program-file.

`DEFMODE { OFF | ON }`

specifies the initial DEFMODE setting (controlling both enabling and propagation of DEFINEs) for the process you are starting, and determines which DEFINEs are propagated to the newly started process. If you do not include the DEFMODE option, the initial DEFMODE setting for the process you are starting is the DEFMODE setting for the current TACL. The DEFMODE option has no effect on the DEFMODE setting for the current TACL.

If you specify DEFMODE OFF, all DEFINEs are disabled for the new process, and no DEFINE is propagated from the current TACL to the new process.

If you specify DEFMODE ON, all DEFINEs are enabled for, and propagated to, the new process.

`EXTSWAP [` *`file-name`* `]`

specifies the name of a file to be used as swap space for the default extended data of the process. The EXTSWAP file must reside on the same node as the program file.

For processes running with pre-D42 software RVUs, omitting this option causes the Kernel-Managed Swap Facility (KMSF) to create the default extended swap file on the volume specified by the SWAP volume in the =_DEFAULTS DEFINE. If there is no SWAP volume in the =_DEFAULTS DEFINE, the operating system chooses one. For more information about the KMSF facility, refer to the *Kernel-Managed Swap Facility (KMSF) Manual*.

For non-native processes running with D42 or later software RVU, omitting this option causes the Kernel-Managed Swap Facility (KMSF) to allocate swap space for the default extended data segment of the process. For more information about the KMSF facility, refer to the *Kernel-Managed Swap Facility (KMSF) Manual*.

For native processes running with D42 or later software RVUs, the specified file-name is ignored because these processes do not need an extended swap file.

If you omit this option, the extended swap file is created on the volume specified by the SWAP volume in the =_DEFAULTS DEFINE. If there is no SWAP volume in the =_DEFAULTS DEFINE, the operating system chooses a volume for the extended swap file.

`GUARANTEEDSWAPSPACE` *`number-of-bytes`*

where *`number-of-bytes`* specifies the size, in bytes, of the space that the process reserves with the Kernel-Managed Swap Facility for swapping. For more information on this facility, refer to the *Kernel-Managed Swap Facility (KMSF) Manual*. The value provided is rounded up by operating-system procedures to a page size boundary that is appropriate for the processor. For more information, see the description of the Z^SPACE^GUARANTEE parameter of the PROCESS_LAUNCH_ procedure in the *Guardian Procedure Calls Reference Manual*.

Note. This option is available when the default configuration settings of the TACL process are changed. To change the settings, set the TACL configuration parameter CONFIGRUN to PROCESSLAUNCH. For more information, see the #SETCONFIGURATION option, CONFIGRUN [PROCESSCREATE | PROCESSLAUNCH] on page 9-348.

`HIGHPIN { ON | OFF }`

specifies the desired PIN range for a new process.

`ON`

specifies that a process will run at a high PIN if the HIGHPIN bit is enabled in the object file (and in the library file, if any) and if a high PIN is available.

`OFF`

specifies that the process runs at a low PIN, regardless of any other considerations.

The default value for HIGHPIN depends on the values of the Binder

HIGHPIN option and the HIGHPIN variable. If you start a TACL process with HIGHPIN OFF, any processes started by the TACL process run at a low PIN by default.

`IN [ file-name | $process-name ]`

is the IN file for the new process. This file or process name is sent to the new process in its startup message. If you do not include the IN option, the new process uses the IN file of the current TACL (usually your home terminal). If you include IN with no name, spaces are sent as the name of the input file. TACL allows the IN file to be a DEFINE name, and passes the DEFINE name to the process being executed. The process is responsible for handling the DEFINE.

`INLINE`

specifies that the process is to be run under control of the INLINE facility (for examples, see the *TACL Programming Guide*). This option also has the effect of the NOWAIT option.

`INSPECT { OFF | ON | SAVEABEND }`

sets the debugging environment for the process being started. ON and SAVEABEND select the Inspect symbolic debugger as the debugger; OFF selects the DEBUG facility. SAVEABEND is the same as ON except that it automatically creates a save file if the program abends (ends abnormally.) The INSPECT option sets the debugging environment for the process you are starting and for any descendants of that process. For more information, see the DEBUG Command on page 8-48, SET INSPECT Command on page 8-193, and SHOW Command on page 8-200 and the *Inspect Manual*.

`INV variable-level [ DYNAMIC [ PROMPT variable-level ]]`

is a variable level whose contents are extracted line by line and passed to the process as the process reads from its IN file. If you include the word DYNAMIC, the process waits for the variable to be refilled if it becomes empty By including the PROMPT option, you can capture prompts in the specified variable level: The most recent prompt string from the process is put into the variable level. For additional information about the use of INV, see Considerations on page 8-163.

`JOBID num`

specifies the new job ID for the new process.

`LIB [ file-name ]`

selects a user library file of object routines that are to be searched before the system library file for satisfying external references in the program being run. If you give the name of a library file, the program uses that library until you select another library file. The library file name is linked to the program file and remains in use for all runs of the program until you specify LIB without a file name. If you do not give a file name, LIB deletes the previous selection.

To run a program file with a user library, you must have write access to the program file; the library file name is written into the object-file header of the program at run time.

To run the program again with the same library, you can omit the LIB parameter. To run the program again with no library (or with a different library), include LIB (or LIB *file-name*).

MAXMAINSTACKSIZE *number-of-bytes*

where *number-of-bytes* specifies the maximum size, in bytes, of the process main stack. The value provided is rounded up by operating-system procedures to a page size boundary that is appropriate for the processor. The specified size cannot exceed 32 megabytes (MB). The default value of 0D indicates that the main stack can grow to 1MB. For most processes, the default value is adequate.

Note. This option is available when the default configuration settings of the TACL process are changed. To change the settings, set the TACL configuration parameter CONFIGRUN to PROCESSLAUNCH. For more information, see the #SETCONFIGURATION option, CONFIGRUN [PROCESSCREATE | PROCESSLAUNCH] on page 9-348.

MAXNATIVEHEAPSIZE *number-of-bytes*

where *number-of-bytes* specifies the maximum size, in bytes, of the process heap. This parameter is valid only for “native” processes (that is, processes that execute RISC code without interpretation or emulation).

The sum of the size of the heap and the size of global data cannot exceed 384 megabytes (MB). The default value of 0D indicates that the heap can grow to the default value of 16 megabytes (MB). The initial heap size of a process is zero bytes. For most processes, the default value is adequate.

Note. This option is available when the default configuration settings of the TACL process are changed. To change the settings, set the TACL configuration parameter CONFIGRUN to PROCESSLAUNCH. For more information, see the #SETCONFIGURATION option, CONFIGRUN [PROCESSCREATE | PROCESSLAUNCH] on page 9-348.

MEM *num-pages*

is the maximum number of virtual data pages to be allocated for the new process. Specify num-pages as an integer in the range 1 through 64. If you omit this option, or if num-pages is less than the compilation-time value, the compilation-time value is used instead.

NAME [$*process-name*]

is the name you are assigning to the new process. Specify $*process-name* as an alphanumeric string of one to five characters (not including the dollar

sign); the first character must be alphabetic. (For network access, the name must be no more than four characters.) If you omit this parameter, the new process is not named and has only a CPU number and process number. If you include NAME with no $*process-name*, TACL generates a name for the new process. The name of the process appears in the destination control table (DCT).

`NOWAIT`

means that TACL does not wait while the program runs but returns a command input prompt after sending the startup message to the new process. If you omit this option, TACL pauses while the program runs.

`OUT [ *list-file* ]`

is the output file of the new process. If you omit OUT *list-file*, the new process uses the OUT file in effect for the current TACL (usually your home terminal). If you include OUT with no *list-file*, spaces are sent as the name of the output file. You cannot specify OUT if you specify OUTV or WINDOW.

TACL allows the OUT file to be a DEFINE name, and passes the DEFINE name to the process being executed. The process is responsible for handling the DEFINE.

`OUTV *var-name*`

is a variable whose indicated level is cleared while retaining its type. Lines are then added to it as the process writes to its OUT file. Prompt strings are not written to the OUT variable. You cannot specify OUTV if you specify OUT or WINDOW. For additional information about the use of OUTV, see Considerations on page 8-163.

`PFS *num-pages*`

is the process file segment size, in 2048-byte pages, for the new process. Specify num-pages as an integer value in the range 64 to 512. If you omit this option, the number of pages is determined by a value in the program object file.

`PRI *priority*`

specifies the execution priority of the new process; processes with higher numbers run first. Specify priority as an integer in the range 1 to 199. If you specify a priority greater than 199, the process runs at priority 199.

If the priority of the TACL process is 1 and *priority* for the new process is not specified, TACL starts the new process at the priority of the TACL process.

If the priority of the TACL process is greater than 1 and *priority* for the new process is not specified, TACL starts the new process at 1 less than the priority of the TACL process.

If a $CMON process exists, it might specify a different priority for the new process, depending on how it has been coded. See the *Guardian Programmer's Guide* for information about $CMON processes.

After a process has been started, the ALTPRI command can be used to alter its priority.

STATUS *variable*

indicates why a process stops. Sets the variable to one of the values STOP, ABEND, CPU, or NET when the process ends. To use STATUS, the TACL process must be named.

SWAP *swap-file*

specifies the name of the file used to hold the virtual data of the process. When a process is running, the system allocates a temporary file on the same volume as the program file for swapping the data stack. When the process terminates, the temporary swap file is automatically purged. If the swap file has a permanent name, however, it is not purged.

With the SWAP parameter, you can:

- Specify a permanent file name-the file contains an image of the data stack when the process terminates.
- Specify a different volume for the swap file (by specifying only a volume name)-this is useful when the program file volume is full or busy. The SWAP option also specifies the default volume for extended data segments; see the *Guardian Programmer's Guide* for more details.
- SWAP can be used in debugging or for improving process performance.

For nonnative processes running with D42 or later software RVUs, the swapfile is not used (or if provided, is ignored). The Kernel-Managed Swap Facility (KMSF) manages swap space, including the file location, for the process. The #PROCESSINFO built-in function always returns "$*volume*.#0" for the SWAP *file-name*. For more information about the KMSF facility, refer to the *Kernel-Managed Swap Facility (KMSF) Manual*.

TERM [*node-name*.]$*terminal-name*

specifies the name of the home terminal (or a DEFINE that contains the name) for the new process. If you omit this option, the new process uses the TACL home terminal. For $*terminal-name*, specify a valid name for a terminal or process: following the dollar sign, specify an alphanumeric string of one to six characters; the first character must be alphabetic. For remote access, you can have no more than five characters after the dollar sign.

`WINDOW [" text "]`

creates a window for the OUT file of the new process. The quotation marks are required; otherwise, TACL returns an error message. This option is for use in an X Windows environment. The home terminal is inherited from the parent TACL process; any I/O to the home terminal is directed to the parent TACL. Output to the home terminal is not displayed if there is a read or write pending on the home terminal. For example, TACL does not display output from the new process while the parent TACL is prompting the home terminal for input. You cannot specify WINDOW if you specify OUT or OUTV. When using the x6530 terminal emulator, you can use text as a parameter list for x6530 configuration. For example:

```
FUP / WINDOW "-name FUP"/
```

causes an X Window to be created for a FUP process. Both the icon and the window banner have the name “FUP.” For additional information, see the *x6530 User’s Guide*.

param-set

is one or more program parameters sent to the new process in the startup message. Leading and trailing spaces are deleted. TACL metacharacters cannot be included unless they are preceded by tildes or are input under control of PLAIN or QUOTED format (see the #INFORMAT Built-In Variable on page 9-196).

Considerations

These considerations apply to the RUN command:

- The RUN command runs object files of type 100, 700, 800, or TACL programs.
- You can specify DEBUG as the debugger with the SET INSPECT OFF command or with the INSPECT OFF option of the RUN command. For more information, see the SET INSPECT Command on page 8-193. For information about INSPECT, see the *Inspect Manual*. For information about DEBUG, see the *Debug Manual*.
- If you are not the super ID, you can debug only those programs whose process accessor IDs match your user ID. For a description of process accessor IDs and process creator IDs, see the *Guardian User’s Guide*.
- Privileged programs can be licensed (by the super ID) for use by users other than the super ID. However, only the super ID can debug privileged programs. If you are not the super ID and you try to use the RUND or RUNV command to debug a licensed program, the program runs without entering a debug state.
- For TNS/R systems, data swap files require 32,000 bytes more than the amount specified in the MEM parameter. For example, if you specify MEM 64, TACL doubles 64,000, rounds up to the next multiple of 4 (resulting in the same number in this case), and adds 32,000 bytes, resulting in 160,000 bytes. Similarly, for a MEM 3 setting TACL doubles 3,000 bytes, rounds up to the next multiple of 4

(8,000 bytes), and adds 32,000, resulting in 40,000 bytes. Ensure that your named swap file has enough extent file space.

- To use either the INV or the OUTV option, your TACL process (the one from which you are starting the new process) must have a process name.
- To use INV dynamically, you must include the NOWAIT option, so that control returns to your TACL process. To send information, wait for the prompt variable.If you plan to wait for more than one prompt, clear the prompt variable prior to the next wait.
- If you include either the IN or the INV option, you cannot use the INLINE option, and the reverse. You cannot use the INLINE option if a process started previously with the INLINE option still exists.
- If you include the INLINE option, TACL waits until the newly started process prompts for the first time; this guarantees that the initial output of the process is available in the #INLINETO variable (if any) when TACL resumes operation.
- When running a process that communicates with TACL (such as by setting IN or OUT to the TACL process name, or by using TACL variables in INV or OUTV, or by using the INLINE feature), be careful to coordinate TACL functions that enable the communication (such as #IN or #OUT) with the matching mechanisms in that process. Deadlock conditions can result if TACL tries to open a process for communication at the same time that process is trying to open TACL for communication.
- When you include the LIB option, the operating system tries to resolve external references to procedures in the program. It searches the library file specified with the LIB option when the program was run. The date and time of the last modification of LIB, as well as the disk address of the library file, are stored in the program file. When you run a program without specifying a library with the LIB option, the operating system compares the disk address and modification date of the actual library file with the information about the library in the program file. If they do not match, the message “LIBRARY FILE CONFLICT” is displayed. This safe-guard prevents you from inadvertently running the wrong version of the library.
- To use LIB, the user who runs a program should have write access to the program file. For example, assume these two users:

  ```
  GROUPA.USER1 GROUPB.USER2
  ```

  and two files (with security CUCU) owned by USER1:

  ```
  $DATA.USER1.PROG
  $DATA.USER1.LIBFILE
  ```

  These two attempts to run program PROG by USER1 succeed:

  ```
  14> RUN PROG
  15> RUN PROG / LIB LIBFILE /
  ```

USER2 (who does not have write access to PROG) can also run PROG, provided the library LIBFILE has not been altered since its last use. If LIBFILE has changed, however, and USER2 enters:

```
12> RUN PROG / LIB LIBFILE /
```

the attempt to run the program fails because USER2 does not have write access to PROG so that external references can be resolved through the changed library file.

- When you give a RUN command and the new process begins execution, TACL pauses unless your RUN command includes the NOWAIT option. If the new process does not take over break ownership, you can activate TACL while the process runs by pressing the BREAK (or interrupt) key. TACL then runs concurrently with the process. You can return TACL to its waiting state with the PAUSE command.
- If you specify the NOWAIT option in your RUN command, TACL returns to the command input mode as soon as the new process reads its startup message. Thus, NOWAIT means you do not have to wait for the new process to finish before you can enter other commands. NOWAIT is especially useful when you start several programs using the IN file-name option. The INLINE option also produces the effect of the NOWAIT option.
- You can use the STATUS option to avoid race conditions. For example, assume a macro runs the same program more than once:

  ```
  RUN laps /INV iv1 DYNAMIC,OUTV ov1,NOWAIT,NAME $z
  STOP $z
  RUN laps /INV iv2 DYNAMIC,OUTV ov2,NOWAIT,NAME $z/
  ```

  The second time the program is run, if TACL receives the OPEN system interprocess message for the second process's IN file before it receives the STOP interprocess message for the first process, the process could receive a "nonexistent file" error when it tries to open its IN file.

  Before starting a second process with the same name as the first, the macro must wait for the STOP command to finish. You can make the macro wait by providing a STATUS variable in the first RUN command, then using the #WAIT built-in function to ensure that TACL receives notification that the first process has stopped before it starts the second one.

  For example:

  ```
  RUN laps /INV iv1 DYNAMIC,OUTV ov1,NOWAIT,NAME $z,STATUS
  zs/
  STOP $z
  #WAIT zs
  RUN laps /INV iv2 DYNAMIC,OUTV ov2,NOWAIT,NAME $z/
  ```

- To run a process on a remote system, specify \ node-name before the name of a program file. For example, this command runs a TEDIT process on the \CHICAGO system:

  ```
  14> \CHICAGO.TEDIT
  ```

Similarly, these commands also run a TEDIT process on the \CHICAGO system, because the current default node name is used for file-name expansion:

```
14> SYSTEM \CHICAGO
15> TEDIT
```

A program file, however, must reside on the system where it runs; the command:

```
14> RUN \DETROIT.MYPROG
```

attempts to run a program file named:

```
\DETROIT. default-volume. default-subvolume.MYPROG
```

If no such program file exists on the remote system, a message is displayed indicating that the file does not exist. See the *Expand Management Programming Manual* for further information on creating remote processes.

- If you specify more than one out-run-option, TACL returns “Option conflicts with another option.”
- If the PFS option is out of range, TACL returns “Expecting a number or an arithmetic expression (Its value must be between 64 and 512 inclusive).”
- Redirection abilities of the OSH command utility can be used. For more information on redirection, see Section 6, Open System Services Shell and Utilities Reference Manual.

These conditions apply to the use of the RUN command for starting TACL processes:

- A TACL process starts in a logged-off state. If, however, Safeguard software authenticates the user and starts the TACL process, that TACL process starts in a logged-on state.
- To run TACL as a server process, set the IN file to $RECEIVE. For more information, see the *TACL Programming Guide*.
- If the IN file is the same as the OUT file and the TACL process is not named, TACL does not set its home terminal.
- The OSH process should not be started with the INLINE option. If you use the INLINE option:
  - TACL will not be able to detect the end of the INLINE process.
  - The output displayed might not be synchronized with the command entered using the inline prefix.

Examples

1. This implicit RUN command runs the text editor program named TEDIT that resides in the file $SYSTEM.SYSTEM.TEDIT:

```
14> TEDIT
```

2. This command runs the program APP1 in the current default subvolume:

```
15> RUN APP1
```

3. This command also runs APP1:

```
16> RUN APP1 / IN APP1IN, OUT APP1OUT, CPU 2, &
16> &NAME $PROC1, NOWAIT /
```

   It also specifies:

   - The file APP1IN as the input file for the program
   - The file APP1OUT as the output file for the program
   - Processor 2 as the processor where the program runs
   - $PROC1 as the name of the process that is created
   - TACL is to redisplay its command prompt without pausing for APP1 to complete execution

SEGINFO Command

Use the SEGINFO command to display a table of information about all the TACL segment files in use by your TACL process.

```
SEGINFO
```

SEGINFO displays its information under this heading line:

```
                              Pgs    Pgs    Bytes   Bytes
Segment  File       Access    Now    Max      Now     Max  %     UC    Directory
```

`Access`	is the access mode: PR means private and SH means shared. If access is shared, the segment file is read-only.
`Pgs Now`	is the number of pages currently allocated for the segment file. This number can increase as needed while your TACL runs, but it is never reduced after being allocated.
`Pgs Max`	is the maximum number of pages that could be allocated.
`Bytes Now`	is the number of bytes currently in use in the segment file.
`Bytes Max`	is the maximum number of bytes that the segment file can hold.
`%`	is the percentage of “Bytes Max” that are in use.
`UC`	is the use count for the segment file. Use count is a count of variables in the segment that are being used by your TACL at this instant. A variable is in use if it is being invoked, is being used for process I/O, is in the use list, is in a pushed use list, is the home directory, or is a pushed home directory. The use count also includes one count for the segment file being attached.
`Directory`	is the full path-name of the directory that contains the names of the variables in the indicated segment file.

For more information about segment files, see Section 6, The TACL Environment.

Example

```
24> SEGINFO

                                  Pgs    Pgs    Bytes     Bytes
Segment File             Access   Now    Max      Now       Max  %  UC   Directory
$EM2.#1242                   PR     8   1024    10584   2097152  0  5    :
$SYSTEM.SYSTEM.TACLSEG       SH    56   1024   111340   2097152  5  3    :UTILS.1
F
$EM2.FURD.MYSEG              SH    76   1024    29676   2097152  6  1    :MYDIR.1
```

SEMSTAT Program

Use the SEMSTAT program to print the Binary Sempahore (BINSEM) usage information and statistics for a process whose ID or process name is provided.

Syntax to invoke the command:

```
SEMSTAT {-procname <name> | -pin <pin> [-brief | -full |
          -wide |-clear_stats]}
```

`<name>` or `<pin>`

specifies the process in the CPU where SEMSTAT is running.

`-brief` or `-full` or `-wide`

specifies how much information is displayed and the format used to display the data.

The `-wide` option displays information in one line to facilitate further processing.

`-clear_stats`

resets all BINSEM counters including the maximum number of contenders.

Note.

- All the above options are case insensitive.
- The SEMSTAT program is provided with the operating system and, if run on that system, works without problems. If the SEMSTAT program is moved to another system that is running an incompatible version of NSK, SEMSTAT exits with an error message.

Security Considerations

The SEMSTAT program is accessible to users with appropriate accessibility rights for a process. The following list describes the accessibility rights for various users:

- Users with the same process access ID as the process of interest. Both GROUP and USER IDs must match.
- Group Managers of the process of interest (that is GROUP,255).
- Processes that are members of the SUPER group (that is SUPER.*).

Examples

The SEMSTAT program prints BINSEM statistics for each BINSEM used by the specified process. The statistics are printed in a tabular format and are categorized as the following heading rows:

`Sem ID`	The BINSEM ID in this process.
`Acquisitions`	The number of times the BINSEM is acquired.
`Tot. Cont.`	The number of times the BINSEM is found locked.
`Mult. Cont.`	The number of times the BINSEM is found locked and contended.
`Cur. Cont.`	The number of processes waiting for the BINSEM.
`Max. Cont.`	The maximum number of contenders for a BINSEM.
`Create Date`	The date when the BINSEM is created.
`Create Time`	The time when the BINSEM is created.
`Serial Number`	The serial number of creation (since CPU load) which uniquely identifies the same semaphore shared by multiple processes.
`Timeouts`	The number of times a timeout occurrs when a process tries to acquire a BINSEM.
`Forced`	The number of times a process stole the BINSEM from an unresponsive process.
`Forsaken`	The number of times a process abandoned ownership of a BINSEM.

Samples

1.Brief Format:

```
semstat /cpu 1/ -pin 843
SEMSTAT utility -- T9050J01 - (01AUG12) - (18JUN12) - (AWT)
(c) Copyright 2011 Hewlett Packard Development Company, L.P.


          --- Binary Semaphore Statistics for Process 1,843 ---
Sem ID      Acquisitions    Tot. Cont.   Mult. Cont.  Cur. Cont.  Max. Cont
-------------------------------------------------------------------------
     0              8225             3             0           0          1
     1              8225             0             0           0          0
     2              8225          7851          7833           0         30
     3              8225             0             0           0          0
     4              8225             0             0           0          0
```

2.Full Format

```
semstat /cpu 1/ -pin 843 -full
SEMSTAT utility -- T9050J01 - (01AUG12) - (18JUN12) - (AWT)
(c) Copyright 2011 Hewlett Packard Development Company, L.P.

          --- Binary Semaphore Statistics for Process 1,843 ---
     Sem ID Acquisitions    Tot. Cont.   Mult. Cont.  Cur. Cont.  Max. Cont
Create Date  Create Time Serial Number      Timeouts      Forced   Forsaken
-------------------------------------------------------------------------
          0         8225             3             0           0          1
 06-26-2012     01:30:14            32             0           0          0
          1         8225             0             0           0          0
 06-26-2012     01:30:14            33             0           0          0
          2         8225          7851          7833           0         30
 06-26-2012     01:30:14            34             0           0          0
          3         8225             0             0           0          0
 06-26-2012     01:30:14            35             0           0          0
          4         8225             0             0           0          0
 06-26-2012     01:30:14            36             0           0          0
```

Note. The default display format is `-brief`.

The following example first displays BINSEM Statistics for Process 1,945, and then displays the cleared statistics after using `-clear_stats` option:

```
SYSTEM STARTUP 49> semstat -pin 945
SEMSTAT utility -- T9050J01 - (01AUG12) - (18JUN12) - (AWT)
(c) Copyright 2011 Hewlett Packard Development Company, L.P.

           --- Binary Semaphore Statistics for Process 1,945 ---
Sem ID      Acquisitions    Tot. Cont.   Mult. Cont.  Cur. Cont.  Max. Cont
-------------------------------------------------------------------------
     0              2593             0             0           0          0
     1              2593             0             0           0          0
     2              2593             0             0           0          0
     3              2593             0             0           0          0
     4              2593          2538          2535           0         31



$SYSTEM STARTUP 50> semstat /cpu 1/ -pin 945 -clear_stat
SEMSTAT utility -- T9050J01 - (01AUG12) - (18JUN12) - (AWT)
(c) Copyright 2011 Hewlett Packard Development Company, L.P.

Statistics counters are cleared



$SYSTEM STARTUP 51> semstat /cpu 1/ -pin 945
SEMSTAT utility -- T9050J01 - (01AUG12) - (18JUN12) - (AWT)
(c) Copyright 2011 Hewlett Packard Development Company, L.P.

           --- Binary Semaphore Statistics for Process 1,945 ---
Sem ID      Acquisitions    Tot. Cont.   Mult. Cont.  Cur. Cont.  Max. Cont
-------------------------------------------------------------------------
     0                 0             0             0           0          0
     1                 0             0             0           0          0
     2                 0             0             0           0          0
     3                 0             0             0           0          0
     4                 0             0             0           0          0
```

SET DEFINE Command

Use the SET DEFINE command to set a value for one or more DEFINE attributes in the working attribute set. Values in the working set determine the values for any attributes you omit from the ADD DEFINE command when you create a DEFINE. For more DEFINE information, see the *Guardian User's Guide*.

```
SET DEFINE { { attribute-spec } | { LIKE define-name } }
  [ , attribute-spec ] ...
```

attribute-spec

assigns a value or a list of values to a DEFINE attribute. For attribute-spec, specify either of these:

```
attribute-name value
attribute-name ( value [ , value ] ... )
```

attribute-name

is the name of a DEFINE attribute that you want to establish in the working attribute set. The valid attribute names and the value associated with each are:

- For a SEARCH DEFINE:

  ```
  RELSUBVOLn subvolume-name
  SUBVOLn subvolume-name
  ```

- For a SORT DEFINE:

  ```
  BLOCK size
  CPU cpu-number
  CPUS { ( cpu-number [ , cpu-number ] ... ) | ALL }
  MODE { AUTOMATIC | MINSPACE | MINTIME }
  NOTCPUS ( cpu-number [ , cpu-number ] ... )
  PRI priority
  PROGRAM file-name
  SCRATCH file-name
  SEGMENT size
  SUBSORTS ( DEFINE-name [ , DEFINE-name ] ... )
  SWAP file-name
  ```

- For a SPOOL DEFINE:

  ```
  BATCHNAME batch-name
  COPIES num
  FORM form-name
  HOLD { ON | OFF }
  HOLDAFTER { ON | OFF }
  LOC [\node-name.] collector[.#group-name[. dest] ]
  MAXPRINTLINES num
  MAXPRINTPAGES num
  OWNER [ group-name.user-name | " group-num, user-num "]
  PAGESIZE num
  ```

```
REPORT report-name
SELPRI num
```

- For a SUBSORT DEFINE:

```
BLOCK size
CPU cpu-number
PRI priority
PROGRAM file-name
SCRATCH file-name
SEGMENT size
SWAP file-name
```

- For a TAPE DEFINE:

```
BLOCKLEN block-length
DENSITY { 800 | 1600 | 6250 }
DEVICE device-name
EBCDIC { IN | OUT | ON | OFF }
EXPIRATION date
FILEID file-name
FILESECT volume-order
FILESEQ file-order
GEN gen-num
LABELS { ANSI | IBM | OMITTED | BYPASS | BACKUP |
IBMBACKUP }
MOUNTMSG " text"
OWNER owner-id
RECFORM { F | U }
RECLEN record-length
REELS volumes
RETENTION days
SYSTEM \node-name
TAPEMODE {STARTSTOP | STREAM }
USE { IN | OUT | EXTEND | OPENFLAG }
VERSION num
VOLUME { vol-id | SCRATCH }
```

value

is a value to be associated with attribute-name. For value, specify a parameter that is valid for the specific attribute. The values available for the various attributes are described in these paragraphs.

LIKE *define-name*

specifies that the working attribute set is to have the same attributes and values as the existing DEFINE named in the LIKE clause. You can modify those attributes or add new attributes with attribute-spec entries that follow the LIKE clause.

Considerations

- There is a special class of DEFINE names that begins with an equal sign plus an underscore (=_). These names are reserved for TACL use only. Do not try to

create DEFINE names that begin with these two characters, except for specific purposes that are described in application product documentation.

- When an error occurs for the SET DEFINE command, no attributes or values are changed in the working attribute set.
- After you have set an attribute value, it persists until you reset it. You can reset an attribute value explicitly with the RESET DEFINE command or with another SET DEFINE command. You can reset a value implicitly by changing the CLASS attribute.
- If you log on from a logged-on TACL process, TACL preserves existing DEFINEs.
- If you start a new TACL process from your existing TACL process, the new TACL process does not inherit existing PARAM values.
- When a backup TACL process takes over, TACL deletes existing DEFINEs.
- The primary attribute-name value specification is the CLASS attribute, which is specified as:

```
CLASS { CATALOG | DEFAULTS | MAP | SEARCH | SORT | SPOOL |
SUBSORT | TAPE }
```

The default is CLASS MAP. The CLASS attribute works as a DEFINE subtype, and the seven classes have different uses:

CATALOG

makes a correlation between a logical catalog and an actual subvolume for NonStop SQL/MP.

DEFAULTS

holds the standard default values of a process (such as the default volume).

MAP

makes a correlation between a logical device and an actual file.

SEARCH

specifies a search list of subvolumes for a program; the SEARCH class is similar in functionality to the TACL #PMSEARCHLIST built-in variable.

SORT and SUBSORT

set parameters for FastSort processes and parallel SORTPROG processes.

SPOOL

sets parameters for the Spooler subsystem.

TAPE

is used for accessing labeled tapes.

The CLASS attribute establishes a different initial working attribute set for each class:

- For CLASS CATALOG, the working attribute set always consists of the SUBVOL attribute only (a required attribute that has no default value):

  SUBVOL *subvol-name*

  For subvol-name, give the name of an existing subvolume; the format is:

  [[*node-name*.]$*volume*.] *subvol*

  The SUBVOL attribute specifies a subvolume to be used as a catalog by NonStop SQL/MP. When you use the DEFINE name in a command, NonStop SQL/MP substitutes the catalog subvolume name for the DEFINE name.

- For CLASS DEFAULTS, the working attribute set consists of the CATALOG, VOLUME, SWAP, LANG, and LC attributes.

  The description of these attributes:

  VOLUME

  specifies the default node, volume, and subvolume names.

  SWAP

  specifies the node and volume names to be used for a swap file.

  CATALOG

  specifies the node, volume, and subvolume names of an SQL catalog.

  LANG

  determines values for all Internationalization (I18N) environment variables in the absence of other LC variables.

  LC_ALL

  determines values of all I18N environment variables and has precedence over all other LC variables and LANG variables.

  LC_COLLATE

  determines how characters are ordered, sorted, grouped, and translated.

  LC_CTYPE

  determines character handling.

LC_MESSAGES

determines how messages and process interactive responses are formatted.

LC_MONETARY

determines how currency representations are formatted.

LC_NUMERIC

determines how numbers are represented.

LC_TIME

determines how dates and times are formatted.

Each of these attributes is case-sensitive and can be up to 256 characters. They can be set to any value, although to be effective, the value must match one of the values supported in the process it is used. Supported values are described in the Internationalization library.

- For CLASS MAP, the working attribute set always consists of only the FILE attribute (a required attribute that has no default value):

  FILE *file-name*

  For file-name, specify a file name; the format is:

  [[[*node-name*.]$*volume*.] *subvol*.] *file-name*

  The FILE attribute specifies the file name to be used in place of MAP DEFINE name. When you use the DEFINE name in a command or procedure call, the file system substitutes the value associated with the FILE name for the DEFINE name. (See the examples for the ADD DEFINE Command on page 8-9.)

  You can use a CLASS MAP DEFINE in TACL wherever a file name is accepted. On a RUN or #NEWPROCESS command, however, the new process must be able to handle a DEFINE name in place of a file name if you include a CLASS MAP DEFINE.

- For CLASS SEARCH, the working attribute set consists of up to 21 attributes named SUBVOL0 through SUBVOL20; the format is:

  SUBVOL*n* *subvolume-name*

  The SUBVOL*n* attribute specifies a subvolume for resolving file names in a search list. To specify multiple subvolumes, use multiple SUBVOL*n* attributes. All attributes are optional.

- For CLASS SORT, the working attribute set consists of the attributes listed in Table 8-4 on page 8-179. All attributes are optional.

- For CLASS SPOOL, the working attribute set consists of the attributes listed in Table 8-5 on page 8-182. The only required attribute is the LOC attribute.
- For CLASS SUBSORT, the working attribute set consists of the attributes listed in Table 8-6 on page 8-184. The only required attribute is the SCRATCH attribute.
- For CLASS TAPE, the working attribute set consists of all the attributes listed in Table 8-8 on page 8-186, as well as any values set or defaulted for those attributes. The only required attribute for a TAPE DEFINE is VOLUME, and then only when you specify USE IN. Table 8-8 describes the CLASS TAPE attributes; those marked with an asterisk (*) correspond to fields in the tape system labels.

- Because the CLASS attribute works as a DEFINE subtype, you should remember these points when you use SET DEFINE:
  - Attributes are set in the order in which they are specified in the SET DEFINE command.
  - Setting the CLASS attribute establishes a new working attribute set that consists of all the attributes associated with that class, each with its initial setting.
  - You cannot set an attribute that is not associated with the current CLASS. For example, if the current CLASS is DEFAULTS, you cannot enter SET DEFINE LABELS IBM.
  - To avoid errors or unexpected results with the SET DEFINE command, set the CLASS attribute first. You can do this in a separate SET DEFINE command, with a LIKE clause in a SET DEFINE command, or as the first attribute in a SET DEFINE command. However, be careful not to specify a LIKE clause in the same SET DEFINE command with a CLASS clause.
- If you enter a SET DEFINE command with a LIKE clause, a later ADD DEFINE command will create a DEFINE identical to the one named in LIKE define-name as long as you do not modify any attributes. These points apply:
  - If the CLASS of LIKE define-name is the same as the current CLASS, all the attributes in the working attribute set are set to the values of define-name. For example, if an attribute has no value in LIKE define-name, that attribute has no value in the working attribute set.
  - If the CLASS of LIKE define-name is different from the current CLASS, a new working attribute set is established, corresponding to the CLASS of define-name, and with attribute values set as in define-name.
- Any attribute specifications that follow a LIKE clause in a SET DEFINE command modify the attribute values established by the LIKE clause.

- The same set of DEFINE attributes can be configured for a generic process through SCF. For the syntax, see the *SCF Reference Manual for the Kernel Subsystem*.

Creating a SORT DEFINE

Table 8-4 describes the attributes that apply to the FastSort subsystem, and the values available for those attributes. See the *FastSort Manual* for a full description of the effects of these attributes. 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 the DEFINE and uses them to set the sort parameters. =_SORT_DEFAULTS is reserved for use as the default SORT DEFINE name.

Table 8-4. SORT DEFINE Attributes (page 1 of 3)

Name and Value	Function
BLOCK *size*	Specifies the size, in bytes, of input and output blocks for a SORTPROG scratch file It can be any multiple of 512 up to 30 KB and it must be large enough to accept the largest input record, rounded up to the nearest even byte, plus 14 bytes overhead. The default is 16 KB.
CPU *cpu-num*	Specifies the number, in the range from 0 through 15, of the processor in which to run a SORTPROG process The default is the same CPU in which FastSort is running.
CPUS { (*cpu-num* [, *cpu-num*] ...) | ALL }	Specifies CPU numbers, in the range from 0 through 15, of processors available for use by subsorts ALL specifies all processors are available.
MODE { AUTOMATIC | MINSPACE | MINTIME }	Specifies FastSort control mode
AUTOMATIC	Minimizes elapsed time by using not more than 50% (90% in parallel sorting) of memory not appropriated by the operating system For files up to 100 KB, FastSort uses an extended memory segment of 64 pages and makes no merge pass; for larger files, FastSort uses enough memory to make only one merge pass.

Table 8-4. SORT DEFINE Attributes (page 2 of 3)

Name and Value	Function
MINSPACE	Limits the size of the extended memory segment to 64 pages (128 KB) For files up to 100 KB, FastSort makes no merge pass (or only one merge pass).
MINTIME	Minimizes elapsed time by using not more than 70% of memory not appropriated by the operating system For files up to 200 KB, FastSort uses an extended memory segment of 64 pages, or as much memory as needed to avoid a merge pass; for larger files, FastSort uses enough memory to make only one merge pass.
NOTCPUS (*cpu-num* [, *cpu-num*] ...)	Specifies CPU numbers, in the range from 0 through 15, of processors not available for use by subsorts
PRI *priority*	Specifies the execution priority, in the range from 1 through 199, for the SORTPROG process See PRI run-option for the RUN[D|V] Command on page 8-156.
PROGRAM *file-name*	Specifies a program file to be run in place of $SYSTEM.SYSTEM.SORTPROG
SCRATCH *file-name*	Specifies the name of a disk file for use as a sort work file If the file already exists, it must be unstructured. A volume name alone is acceptable. The default is a temporary file on $SYSTEM.

Table 8-4. SORT DEFINE Attributes (page 3 of 3)

Name and Value	Function
SEGMENT *size*	Specifies the size, in pages, of an extended memory segment for FastSort to use; it must be at least 64%, but not over 90% of the memory not appropriated by the operating system If you specify SEGMENT, you must omit MODE. The default is the same as MODE AUTOMATIC.
SUBSORTS (DEFINE-*name* [, DEFINE-*name*] ...)	Specifies one or more DEFINE names, of class SUBSORT, available for use by this DEFINE Those DEFINEs must exist, but are not checked for validity until the sort begins.
SWAP *file-name*	Specifies a swap file for use by the extended memory segment If the file already exists, it must be unstructured. A volume name alone is acceptable if it is on the local system. The default is a temporary file on the same volume as the scratch file, if that file is local, or a temporary file on $SYSTEM, if it is not.

Creating a SPOOL DEFINE

Table 8-5 describes the attributes that apply to the Spooler subsystem, and the values available for those attributes. The only required SPOOL attribute (other than CLASS) is LOC.

Table 8-5. SPOOL DEFINE Attributes (page 1 of 2)

Name and Value	Function
BATCHNAME *batch-name*	Specifies a batch name for a job
	It can be from 1 to 31 characters in length, and can contain hyphens as well as alphanumeric characters. It must not begin or end with a hyphen, and must contain at least one letter. The default is all spaces.
COPIES *num*	Specifies the number of copies, in the range from 1 through 32767, to be printed
	The default is 1.
FORM *form-name*	Specifies a form name for jobs created by the DEFINE, denoting requirements (such as special paper) associated with a job
	It is a string from 1 to 16 characters in length. The default is all spaces.
HOLD { ON | OFF }	Sets the hold flag for jobs created by any process using the DEFINE
	The default is OFF.
HOLDAFTER { ON | *OFF* }	Sets hold-after-printing flag for jobs created by any process using the DEFINE
	The default is OFF (delete job from spooler after printing).
LOC [*node-name*.] $*collector* [.*group-name*[.*dest*]]	Specifies a spooler location to which jobs are to be sent
MAXPRINTLINES *num*	Specifies the maximum number of lines per job, in the range from 1 through 65534
	Exceeding the limit causes a “file full” error. The default is no limit.
MAXPRINTPAGES *num*	Specifies the maximum number of pages per job, in the range from 1 through 65534
	Exceeding the limit causes a “file full” error. The default is no limit.

Table 8-5. SPOOL DEFINE Attributes (page 2 of 2)

Name and Value	Function
OWNER { *group-name.user-name* }| { *group-num.user-num* }	Specifies the owner of all jobs created by any process using the DEFINE The numeric form of user ID must be enclosed in quotes. The default is the user ID of the spooler request initiator.
PAGESIZE *num*	Specifies the number of lines per page, in the range from 1 through 32767, to be used by PERUSE when it performs a LIST or PAGE command The de-fault is a size specified by the creating process or, if no such process exists, the default page size of PERUSE.
REPORT *report-name*	Specifies a report name, 1 to 16 characters in length, to be printed in the job header created by any process using the DEFINE It can contain any alphanumeric characters, but must begin with a letter. If it contains spaces, it must be enclosed in quotes. The default is the group-name.user-name of the owner.
SELPRI *num*	Specifies the selection priority, in the range from 0 through 7 (0 is lowest), of jobs created using the DEFINE The default is 4.

Creating a SUBSORT DEFINE

Table 8-6 lists the attributes that apply to parallel sorts run under the FastSort subsystem and the values available for those attributes. See the *FastSort Manual* for a full description of the effects of these attributes. The only required attribute (other than CLASS) is SCRATCH.

Table 8-6. SUBSORT DEFINE Attributes

Name and Value	Function
BLOCK *size*	Specifies the size, in bytes, of input and output blocks for a subsort scratch file It can be any multiple of 512 up to 30 KB and it must be large enough to accept the largest input record, rounded up to the nearest even byte, plus 14 bytes overhead. The default is 16 KB.
CPU *cpu-num*	Specifies the number, in the range from 0 through 15, of the processor in which to run the subsort process The default is the same CPU in which the primary disk process for the scratch file's volume is running.
PRI *priority*	Specifies the execution priority, in the range from 1 through 199, for the subsort process See the RUN[D|V] Command on page 8-156.
PROGRAM *file-name*	Specifies a program file to be run in place of $SYSTEM.SYSTEM.SORTPROG
SCRATCH *file-name*	Specifies the name of a disk file for use as a sort work file If the file already exists, it must be unstructured. A volume name alone is acceptable. This attribute is required.
SEGMENT *size*	Specifies the size, in pages, of an extended memory segment for the subsort to use It must be at least 64%, but not over 90% of the memory not appropriated by the operating system. The default is 64 pages.
SWAP *file-name*	Specifies a swap file for use by the extended memory segment If the file already exists, it must be unstructured. A volume name alone is acceptable if it is on the local system. The default is a temporary file on the same volume as the scratch file, if that file is local, or a temporary file on $SYSTEM, if it is not.

Creating a TAPE DEFINE

Attribute values for a TAPE DEFINE must meet certain consistency rules, shown in Table 8-7. The specific rules for particular attributes are listed in Table 8-8 on page 8-186.

You can display all the attributes that are currently set or defaulted with the SHOW DEFINE * command. This command also checks these attributes for consistency and returns the check number of the first consistency check that fails.

Table 8-7. TAPE DEFINE Attribute Consistency Rules

Check Number	Description
001	You can specify either RETENTION or EXPIRATION, but not both.
002	If you specify USE IN or EXTEND, you must include VOLUME and specify LABELS ANSI or LABELS IBM. If you specify REELS, the value must equal the number of volumes specified in VOLUME.
003	If you specify VOLUME, you must also specify LABELS ANSI, LABELS IBM, or LABELS IBMBACKUP. If you specify LABELS, you must also specify VOLUME.
004	If you specify LABELS ANSI, you must not specify the EBCDIC attribute, and the reverse.
005	If you specify RECFORM F, you must specify a BLOCKLEN that is a multiple of RECLEN.
006	If you specify DEVICE, you cannot specify SYSTEM in the same DEFINE, and the reverse.
007	If you specify LABELS BYPASS or LABELS OMITTED, you must specify a DEVICE; you must not specify any of these attributes: BLOCKLEN, EBCDIC, EXPIRATION, FILEID, FILESECT, FILESEQ, GEN, OWNER, RECFORM, RECLEN, REELS, RETENTION, SYSTEM, USE, VERSION, VOLUME.
008	If you specify VOLUME SCRATCH, you must not specify USE IN or USE EXTEND.
009	If you specify LABELS IBM or LABELS IBMBACKUP, you must specify a FILEID.
010	If RECLEN is less than 24, you must specify a BLOCKLEN.
011	If you specify LABELS IBMBACKUP, the system you specify in the SYSTEM or DEVICE attribute must have an operating system RVU of C20 or later.

Table 8-8. TAPE DEFINE Attributes (page 1 of 4)

Name and Value	Function
* BLOCKLEN *block-length*	Specifies the data block size, in bytes, in a tape file The default is that the tape process does not check block length (input files). If RECFORM is F, BLOCKLEN must be a multiple of RECLEN.
DENSITY {800 | 1600 | 6250}	Specifies the tape density in bits per inch The specified density appears in mount messages sent to the operator. The default is the current setting of the tape drive.
DEVICE $*device-name*	Specifies the name of the tape device where the tape file is to be mounted If you specify a tape drive on a remote system, the system must be a node on your network. If you omit both DEVICE and SYSTEM attributes, tapes must be mounted on the local system. If you specify DEVICE, you must omit SYSTEM.
EBCDIC {IN | OUT | ON | OFF}	Specifies whether data is to be translated when processing an IBM tape (9-track IBM tape labels are always in EBCDIC) IN: Data records read from the tape file are translated from EBCDIC to ASCII. OUT: Data records written to tape are translated from ASCII to EBCDIC. ON: Both IN and OUT; this is the default for IBM tapes. OFF: Data records are not translated.
* EXPIRATION *date*	Specifies the expiry date for this tape file (the first date on which the file can be overwritten) Specify month, day, and year, such as DEC311990. If you specify EXPIRATION, you must omit RETENTION.
* FILEID *file-name*	* FILEID file-name
* FILESECT *volume-order*	Specifies the position of this volume within a multivolume file being created at the same time Specify an integer in the range 0001 to 9999 to indicate the relative position of the volume. This number is always 0001 for a single-volume file.
* FILESEQ *file-order*	Specifies the position of this tape file in a multifile volume Specify an integer in the range 0001 to 9999 to indicate the position of the file in the volume. This number is always 0001 in a single-file organization.

Table 8-8. TAPE DEFINE Attributes (page 2 of 4)

Name and Value	Function
* GEN *gen-number*	Indicates that this file is part of a generation group
	Specify an integer in the range 0001 to 9999 to indicate the absolute generation number. The default is 0001.

LABELS
{
ANSI |
IBM |
OMITTED |
BYPASS |
BACKUP |
IBMBACKUP
}
	Specifies the type of tape and, for labeled tapes, the label processing mode to be used

	If you specify LABELS, you must also specify VOLUME.
	ANSI: The tape file is on an ANSI-standard labeled tape, and the system is to perform standard label processing on the file (LP mode).
	IBM: The tape file is on an IBM-standard labeled tape, and the system is to perform standard label processing on the file (LP mode). Any DEFINE that includes LABELS IBM must also include RECFORM.
	OMITTED: The tape file is not on a standard labeled tape, and the system does no label processing except to check that tape is not a standard labeled tape (NL mode). If you specify LABELS OMITTED, you must also include DEVICE.
	BYPASS: The system does no label processing and does not check whether tape is labeled (BLP mode). If you specify LABELS BYPASS, you must also include DEVICE.
	BACKUP: The tape is used for labeled-tape BACKUP or RESTORE operations. (LABELS BYPASS is also acceptable for tapes to be read in RESTORE operations, but not for tapes written by BACKUP operations.)
	IBMBACKUP: Indicates that the tape label is in IBM-MVS format used by BACKCOPY, BACKUP, and RESTORE.
* OWNER *owner-id*	Identifies the owner ID in the VOL1 label for IBM labeled tapes only (LABELS must be IBM or ANSI)
	Specify any identifying name or code from 1 to 14 characters long.
MOUNTMSG "*text*"	Specifies an additional mount message to be displayed along with the system mount message or the drive usage request printed when this DEFINE is opened
	Specify a quoted or unquoted character string of up to 80 characters. Include information such as the length and urgency of tape job.

Table 8-8. TAPE DEFINE Attributes (page 3 of 4)

Name and Value	Function
* RECFORM { F | U }	Specifies the record format If you include LABELS IBM, you must also specify RECFORM. F: Indicates fixed-length records (the default for ANSI tapes). U: Indicates undefined length (the default for IBM tapes). For input files, BLOCKLEN, RECFORM, and RECLEN values are not checked for consistency. But if you enter any of these attributes, the corresponding field in the label must match, or the tape will be rejected. For output files, if you do not specify BLOCKLEN, RECFORM, or RECLEN for an IBM-standard labeled tape, the open is rejected. For an ANSI-standard labeled tape, these defaults are assumed: RECFORM U RECLEN 0 BLOCKLEN as configured for device If you specify BLOCKLEN, RECFORM, and RECLEN for a labeled-tape output file, label fields are generated with those values.
* RECLEN *record-length*	Specifies the record length of the tape file For ANSI-standard tapes with RECFORM F, the default RECLEN is the value configured for the device by SYSGEN; if RECFORM is U, the default RECLEN is 0. The record length can be in the range from 0 through 32767.
REELS *volumes*	Specifies the number of volumes in a multivolume file It is mandatory to specify this for any multivolume input file. Specify an integer in the range from 0 through 255. The default is 1.
* RETENTION *days*	Specifies the retention period for the tape file Specify an integer indicating number of days to retain the tape file. This value is translated to an expiration date when labels are written on the tape. The expiration date prevents overwriting the tape contents. The default is zero (the file expires immediately). If you specify RETENTION, you must omit EXPIRATION. Specify an integer in the range from 1 through 32767.
SYSTEM *node-name*	Gives the name of the system that contains the tape drive specified in this DEFINE If you include SYSTEM, you must omit DEVICE.

Table 8-8. TAPE DEFINE Attributes (page 4 of 4)

Name and Value	Function
TAPEMODE { STARTSTOP | STREAM }	Specifies the operating mode for a cartridge tape drive; for other types of drives, this attribute is ignored If you specify TAPEMODE, BLOCKSIZE must be greater than the default of 2 to speed the tape writing process and to produce a more compact tape. The default is STARTSTOP.
USE { IN | OUT | EXTEND | OPENFLAG }	Specifies how the tape file is to be used IN: The file is to be read from. OUT: The tape is to be written to. EXTEND: Data is to be appended to the tape file. OPENFLAG: Uses the type of access indicated by the access flag of the OPEN call (it must be either Read or Write; Read/Write becomes Read).
* VERSION *number*	Indicates a version within one generation Specify an integer in the range 00 to 99. The default is 00.
VOLUME { *vol-id* | SCRATCH }	Specifies one or more tape volume IDs, or specifies that any scratch tape is acceptable for a labeled BACKUP or ANSI tape Specify a unique one-byte to six-byte identification code assigned to the volume; for a multivolume file, enclose the list of volume IDs in parentheses. SCRATCH represents a scratch tape. If you specify USE IN, you must include a VOLUME attribute; otherwise, its value is SCRATCH. If you specify a VOLUME attribute, you must specify LABELS ANSI, LABELS IBM, or LABELS BACKUP. If you specify VOLUME SCRATCH, you cannot specify USE IN or USE EXTEND. **Note**: The maximum number of tape volumes that can be named in a CLASS TAPE DEFINE is 61.

* Attributes marked with an asterisk have corresponding fields in the tape system labels. See the description of tape label formats in the *Guardian User's Guide*.

Examples

1. This command establishes a working attribute set that describes a tape file residing on three ANSI standard tape volumes (1, 2, and 3). This file is to be read (USE IN), and the system is to do standard label processing:

```
13> SET DEFINE CLASS TAPE, LABELS ANSI, VOLUME (1,2,3),&
13> &REELS 3, USE IN
```

2. In this example, a SET DEFINE command establishes a working attribute that contains the attributes common to two DEFINEs that are to be created. Each is a

CLASS TAPE DEFINE that describes a tape file residing on volume 30 of an ANSI standard labeled tape mounted on tape drive $TAPE2.

Next, a SHOW DEFINE command displays the status of the current working attribute set. Finally, two ADD DEFINE commands create the DEFINEs and set the attributes that are unique to each DEFINE, which in this case are the file names MAYRCDS and JUNRCDS:

```
14> SET DEFINE CLASS TAPE, LABELS ANSI, FILEID empty,&
14> &DEVICE $tape2, VOLUME 30

15> SHOW DEFINE
    CLASS                TAPE
    VOLUME               30
    LABELS               ANSI
    FILEID               empty
    DEVICE              $TAPE2

16> ADD DEFINE =one, FILEID mayrcds
17> ADD DEFINE =two, FILEID junrcds

18> INFO DEFINE (=one, =two), DETAIL

    DEFINE NAME           =one
    CLASS                 TAPE
    VOLUME                30
    LABELS                ANSI
    FILEID                mayrcds
    DEVICE               $TAPE2

    DEFINE NAME           =two
    CLASS                 TAPE
    VOLUME                30
    LABELS                ANSI
    FILEID                junrcds
    DEVICE               $TAPE2
```

SET DEFMODE Command

Use the SET DEFMODE command to enable or disable the use of DEFINEs in the current TACL process.

```
SET DEFMODE { ON | OFF }
```

ON

enables the use of DEFINEs in, and the propagation of DEFINEs from, the current TACL process. This means that unless you change the DEFMODE setting in a RUN command, the process started by that command has an initial DEFMODE setting of ON, and all DEFINEs are propagated to that new process.

OFF

disables the use of all DEFINES in the current TACL and the propagation of DEFINEs from it. When DEFMODE is OFF, the current TACL cannot use any DEFINEs. Unless you change the DEFMODE setting in a RUN command, the process started by that command has an initial DEFMODE setting of OFF, and no DEFINEs are propagated to that new process.

Consideration

DEFMODE is always set ON when TACL is first started, and whenever you log on from the logged-off state.

SET HIGHPIN Command

Use the SET HIGHPIN command to establish the default PIN range for processes started by the current TACL when there is no HIGHPIN directive on a RUN command or #NEWPROCESS call.

```
SET HIGHPIN { ON | OFF }
```

`ON`

specifies that a process will run at a high PIN if the HIGHPIN bit is enabled in the object file (and in the library file, if any) and if a high PIN is available. ON is the default value for HIGHPIN.

`OFF`

specifies that processes run at a low PIN, regardless of any other considerations.

Considerations

- This command sets the built-in variable #HIGHPIN. Like other built-in variables, #HIGHPIN can be set, expanded, pushed, and popped.
- Use SHOW HIGHPIN to display the current value of #HIGHPIN interactively, or type #HIGHPIN to expand the variable in a routine.

SET INSPECT Command

Use the SET INSPECT command to establish default debugging conditions for processes started by the current TACL.

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

OFF

disables the Inspect symbolic debugger and selects the Debug program as the default debugger. (The Debug program is the system default debugging utility.) The Debug program then prompts for input when any process created by the current TACL (or any of its descendants) enters the debug state.

ON

selects the Inspect symbolic debugger as the default debugger for all programs started by the current TACL. The Inspect debugger then prompts for input when any process created by the current TACL (or by any descendant of the current TACL) enters the debug state.

SAVEABEND

establishes the Inspect symbolic debugger as the default debugger and automatically creates a save file if the program ends abnormally.

Considerations

- The Inspect product is a symbolic debugger. It allows you to control running processes and SCREEN COBOL programs and to examine memory and modify data values-all with commands that use your source language.

  In addition to the source-language commands, the Inspect debugger supports machine-level commands for maximum debugging flexibility. (For more information, see the *Inspect Manual*. See also DEBUG Command on page 8-48, RUN[D|V] Command on page 8-156, and SHOW Command on page 8-200.)

- Your selection of the Inspect debugger as the default debugger is effective until you enter another SET INSPECT command or until you log off. After you log off, the Debug program once again becomes the default debugger. However, if you enter a SET INSPECT command and log on again without logging off, Inspect remains the default debugger.

SETPROMPT Command

Use the SETPROMPT command to change the TACL prompt. By default, TACL prompts with a history number and a greater-than sign (>) followed by a space.

```
SETPROMPT { SUBVOL | VOLUME | BOTH | NONE }
```

SUBVOL

displays the current subvolume, followed by the command number, a greater-than sign, and a space.

VOLUME

displays the current volume, followed by the command number, a greater-than sign, and a space.

BOTH

displays the current volume and subvolume, followed by the command number, a greater-than sign, and a space.

NONE

displays the command number, a greater-than sign, and a space. NONE is the default.

Examples

1. This example illustrates how to set your prompt to your current subvolume:

```
12> SETPROMPT SUBVOL
BOOK 13>
```

2. You can also set your prompt to both your volume and subvolume by entering:

```
BOOK 13> SETPROMPT BOTH
$STEIN BOOK 14>
```

SET SWAP Command

Use the SET SWAP command to select the swap volume for all subsequent RUN commands, unless a swap volume is explicitly specified in the RUN command.

```
SET SWAP [ $volume-name ]
```

$volume-name

is the name of the volume used to hold virtual data during memory swaps of the user data stack during process execution.

Considerations

- To clear any swap volume previously set, use the SET SWAP command without the *$volume-name* parameter.
- Any RUN command that explicitly specifies a swap volume overrides the previous SET SWAP *$volume-name* command.
- If no SET SWAP command has been issued, and no swap volume is specified in a RUN command, the swap volume defaults to the volume in which the program is stored.
- This command alters the SWAP attribute of the =_DEFAULTS DEFINE.

SETTIME Command (Super-Group Only)

Use the SETTIME command to set the date and time-of-day clock for the system. You normally use SETTIME after you cold load the first processor from disk, but before you load the rest of the system using the RELOAD command. You can also use SETTIME to reset the system clocks after a power failure (the interval clock in a processor module stops when power is interrupted). To use the SETTIME command, you must have a group ID of 255.

```
SETTIME { month day | day month } year , hour: min[: sec] {
GMT | LST | LCT }
```

month

is the name of the month. You must give at least the first three letters of the name of the month. You can use uppercase or lowercase for month.

day

is the day of the month, specified as an integer in the range from 1 to 31, inclusive.

year

is the 4-digit calendar year, from 1975 through 9999.

hour

is the hour of the day, specified as an integer in the range from 0 to 23, inclusive.

min

is the minute of the hour, specified as an integer in the range from 0 to 59, inclusive.

sec

is the second of the minute, specified as an optional integer in the range from 0 to 59, inclusive. If you omit sec, 0 is assumed.

GMT

is Greenwich mean time.

LST

is local standard time.

LCT

is local civil time (LST corrected for daylight-saving time). LCT is the default.

Considerations

- The valid date range is from 01 January 1975 0:00:00.000000 to 31 December 4000 23:59:59.999999.
- If you execute this command while a system monitoring measurement is in process by the XRAY facility, invalid measurements result.
- The CONVERTTIMESTAMP system procedure is invoked while setting the system clock. See #CONVERTTIMESTAMP Built-In Function on page 9-79 for descriptions of the error messages that CONVERTTIMESTAMP could display.
- SETTIME calls the #SETSYSTEMCLOCK built-in function, which in turn calls the SETSYSTEMCLOCK system procedure. If you use SETTIME to set the system clock forward or backward two minutes or less, the system adjusts the clock in small increments rather than setting it to the new time. Adjusting the clock forward two minutes takes about 33 hours. Adjusting the clock back two minutes takes about 14 days.

  If you issue two SETTIME commands in less than ten seconds, the system stops any ongoing adjustment and sets the clock to the value specified in the second call.
- The valid date range is from 01 January 1975 0:00:00.000000 to 31 December 9999 23:59:59.999999. If the system does not support the maximum value for *year* (9999), the SETTIME command returns the error message:

```
Invalid date and/or time format.
```

Example

1. To set the system clock to 8:01 p.m. on July 9, 2004, enter:

```
34> SETTIME JUL 9 2004, 20:01
```

2. You can see the new time by entering the TIME command:

```
35> TIME
July 09, 2004 20:01:ss
```

SET VARIABLE Command

Use the SET VARIABLE command to change the contents of a variable level or built-in variable. The syntax and action of the SET VARIABLE command are the same as the #SET built-in function.

There are two forms of the SET VARIABLE command:

```
SET VARIABLE [ / option [ , option ] / ] variable-level
  [ text ]
SET VARIABLE built-in-variable [ built-in-text ]
```

option

is either of these:

IN *file-name*

specifies that the named file is to be read into the variable level. If this option is present, text is not allowed.

TYPE *type-name*

specifies the type of variable level being set.

type-name

is one of these:

ALIAS

specifies that variable-level is an alias. This implies that text is the name of a variable level or a file.

DELTA

specifies that variable-level is a #DELTA command variable. This implies that text must be #DELTA commands.

DIRECTORY

specifies that variable-level is a directory. If you omit text, this option clears variable-level and establishes an empty directory. If you include text, it must have a specific form:

mode *file-name*

where mode is either PRIVATE or SHARED, and file-name is the name of an existing segment file (code 440); the segment file must not reside on a remote system. This form of the SET VARIABLE command associates a directory variable with a segment file in the same way as an ATTACHSEG command.

MACRO

specifies that text is a TACL macro.

ROUTINE

specifies that text is a TACL routine.

TEXT

specifies that text is simply text (it has no special meaning to TACL).

variable-level

is the name of an existing variable level, of the form:

variable-name[. *level-num*]

If you omit . level-num, the top level of the variable is assumed.

text

is the new contents of the variable level. If the IN option is supplied, you cannot specify text.

built-in-variable

is the name of a built-in variable.

built-in-text

is the new value for the built-in variable.

Considerations

- The syntax and operation of the SET VARIABLE command is the same as that of the #SET built-in function.
- The SET VARIABLE command replaces the current contents of variable-level with the specified text or, if you use the IN option, the contents of the specified file. Unless you specify a TYPE option, the variable type remains the same.
- You cannot use a text string with the IN option, nor can you use the IN option with a built-in variable.
- The SET VARIABLE command cannot put leading or trailing spaces into a variable level.

Example

This example illustrates the use of the SET VARIABLE command. This command sets a variable level named VARA, of type MACRO, to the text *,USER SUPPORT.ALICE:

```
25> SET VARIABLE / TYPE MACRO / vara *,USER SUPPORT.ALICE
```

SHOW Command

Use the SHOW command to display the values of attributes set with the SET command.

```
SHOW [ / OUT list-file / ] [ attribute [ , attribute ] ... ]
```

`OUT list-file`

specifies a device, or a sequential file accessible to the sequential I/O (SIO) facility, that is to receive a listing of command output. If you omit this option, TACL writes the listing to its current OUT file.

If you specify an OUT file that does not exist, TACL creates an EDIT file named list-file. If you specify an OUT file that already exists, TACL appends the information to the end of the file.

attribute

is any attribute controlled by the SET command. Currently, these attributes are controlled by SET:

`DEFMODE`

is the enable mode (ON or OFF) for DEFINEs. For more information, see the RUN[D|V] Command on page 8-156 and SET DEFINE Command on page 8-173.

`INSPECT`

is the name of the alternate symbolic debugging utility. See also the DEBUG Command on page 8-48, the SET INSPECT Command on page 8-193, the RUN[D|V] Command on page 8-156, and the *Inspect Manual*.

`HIGHPIN`

is used to establish the default PIN range for processes started by the current TACL when there is no HIGHPIN directive on a RUN command or #NEWPROCESS call.

SWAP

is the volume that holds virtual data during memory swaps of the user data stack.

Consideration

Entering SHOW without specifying any attributes causes all SET attributes to be displayed.

Examples

1. This command displays the status of the INSPECT and SWAP attributes:

```
13> SHOW SWAP,INSPECT

        Swap $RALPH
        Inspect ON
```

2. To see all attributes currently set, enter:

```
14> SHOW
        Defmode ON
        Highpin ON
        Inspect ON
        Swap
```

The display shows that DEBUG is the debugging utility in effect, DEFMODE is set ON, and that no swap volume has been set.

SHOW DEFINE Command

Use the SHOW DEFINE command to show the value associated with a specific DEFINE attribute, to show all attribute values that are currently set or defaulted, or to show all attributes in the current working set (that is, all the attributes associated with the current CLASS) and the current value of each. For more information about DEFINEs, see Section 5, Statements and Programs.

```
SHOW [ / OUT list-file / ] DEFINE [ attribute-name | * ]
```

`OUT` *`list-file`*

specifies a device, or a sequential file accessible to the sequential I/O (SIO) facility, that is to receive the output from SHOW DEFINE. The listing includes line numbers of the new lines in the destination variable. If you omit this option, TACL writes the listing to its current OUT file.

If you specify an OUT file that does not exist, TACL creates an EDIT file named list-file. If you specify an OUT file that already exists, TACL appends the information to the end of the file.

`attribute-name`

displays the name and current value of the specified attribute-name, a valid name for a DEFINE attribute. Valid attribute names are described under the SET DEFINE command. A required attribute that has no current value is displayed with ?? as its value.

`*`

displays all attribute names and current values for the working attribute set (all attributes associated with the current class); optional attributes that have no current value are listed with a blank value.

Considerations

- Entering SHOW DEFINE with no parameter produces a display of the names and current values of all attributes that are currently set or that have default values. Required attributes that have no current value are listed with ?? as the value. Optional attributes that have no current value are not listed. Attributes whose values violate the consistency rules (see Table 8-7 on page 8-185) are flagged with an asterisk (*).
- The SHOW DEFINE command checks for consistency among the attributes in the working attribute set. If the attributes are inconsistent (that is, if at least one has a value that conflicts with that of another attribute), the inconsistent attribute is flagged with an asterisk, and a warning message is displayed; for example:

```
81> SHOW DEFINE
    CLASS TAPE
* VOLUME 1265
```

```
* LABELS OMITTED
Current attribute set is inconsistent, check number 3
```

If the attributes are incomplete (that is, if a required attribute is missing), a warning message is displayed, and the value for the missing attribute is displayed as ??. For example:

```
87> SHOW DEFINE
    CLASS MAP
    FILE ??
Current attribute set is incomplete
```

TACL returns an error if you specify an attribute that is not a member of the working attribute set (an attribute that is not associated with the current CLASS).

To obtain additional error information, use #ERRORNUMBERS.

Examples

1. To display the value currently set for the DEVICE attribute, enter:

```
92> SHOW DEFINE DEVICE
    DEVICE $TAPE
```

2. This SHOW DEFINE command displays a working attribute set that specifies a CLASS TAPE DEFINE:

```
95> SHOW DEFINE
CLASS TAPE
VOLUME ( 25436, 75444, 23121 )
LABELS IBM
USE IN
```

3. This command displays all working attributes and the value for each:

```
98> SHOW DEFINE *
CLASS TAPE
VOLUME ( 25436, 75444, 23121 )
LABELS IBM
REELS
OWNER
FILESECT
FILESEQ
FILEID
RETENTION
EXPIRATION
GEN
VERSION
RECFORM
BLOCKLEN
RECLEN
DENSITY
RECLEN
USE IN
DEVICE
EBCDIC
```

```
MOUNTMSG
SYSTEM
```

SINK Command

Use the SINK command to invoke a function but discard its result. SINK can discard nonnumeric results as well as numeric ones.

```
SINK [ text ]
```

Consideration

Use of the SINK command discards error indications returned by a function. Therefore, the use of SINK is not recommended unless you do not need to know if an error occurred. For example, if you want to purge a file but do not care if you try to purge the file and the file does not exist, use SINK to discard any error that might occur.

Example

This command loads the TACL library file MYMACS into memory, but does not display the list of variables that were modified:

```
14> SINK [#LOAD mymacs]
```

STATUS Command

Use the STATUS command to display information about one or more running processes. The STATUS command is an alias for the #XSTATUS built-in function.

```
STATUS [ / OUT list-file / ] [ range ] [ , condition ] ...
  [ , DETAIL ] [ , STOP ] [ , FORCED ]
```

OUT *list-file*

specifies a device, or a sequential file accessible to the sequential I/O (SIO) facility, that is to receive the STATUS output. If you omit the OUT option, the STATUS listing goes to the OUT file in effect for the current TACL (usually the home terminal).

If you specify an OUT file that does not exist, TACL creates an EDIT file named *list-file*. If you specify an OUT file that already exists, TACL appends the status information to the end of the file.

range

is any of these:

```
[\node-name.] cpu,pin
[\node-name.] cpu-number
[\node-name.]$process-name
[\node-name.]*
```

node-name

requests the status of all specified processes running in \ node-name.

cpu,pin

requests the status of a particular process.

cpu-number

requests the status of all processes running in a particular CPU.

$*process-name*

requests the status of a particular named process or process pair.

*

requests the status of processes running in all CPUs.

If you omit range, STATUS reports on the last process started by the current TACL, or for which TACL last paused, if that process is still running. is any one of these:

```
GMOMJOBID $process-name. num
PRI [ priority ]
PROG [ program-file-name | file-name-template ]
```

```
TERM [ $terminal-name ]
USER [ user-id ]
```

`GMOMJOBID $process-name.num`

specifies processes with the given job ancestor ID. *num* is a signed integer.

`LOADED [loaded-file-name | file-name-template]`

is used to get the information about processes that are using given loaded files. The same functionality can be used with built-in #XSTATUS.

LOADED specifies processes using the given loaded file name. If you omit the loaded file specification, LOADED defaults to the program file name of the current TACL.

You can use file-name-template characters in any field of the file specification except the system field. The template characters are:

*	Matches zero or more characters
?	Matches a single character

Template characters cannot match a volume identifier($) of a field separator(.).

`PRI [ priority ]`

specifies processes whose execution priority is less than or equal to the priority given. If you omit priority, STATUS reports on processes whose priorities are less than that of the current TACL.

`PROG [ program-file-name | file-name-template ]`

specifies processes with the given program file name. If you omit the program file specification, PROG defaults to the program file name of the current TACL.

You can use file-name-template characters in any field of the file specification except the system field. The template characters are:

*	Matches zero or more characters
?	Matches a single character

Template characters cannot match a volume identifier ($) or a field separator (.).

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

specifies processes running on a given terminal. If you omit $ terminal-name, STATUS reports on processes running on the home terminal of the current TACL process. If you omit node-name, the STATUS command uses the system specified in range or, if not specified in either argument, uses the default system.

`USER [ ident ]`

specifies processes created by a particular user, where ident is either group-name.user-name or group-id, user-id. If you include USER without ident, STATUS reports on processes whose creator accessor ID matches your user ID.

If you specify more than one condition, STATUS reports on all processes that satisfy all the conditions.

`DETAIL`

gives a detailed display of process status.

`STOP`

specifies that TACL is to try to stop each process for which information is displayed (except the TACL process issuing the command), subject to the normal rules governing which processes you are allowed to stop. TACL displays a confirmation message asking if you want to stop the processes specified by the command line. TACL does not indicate whether the STOP option succeeds or fails.

`FORCED`

forces TACL to stop the processes specified by the command line. In this case, TACL will not prompt you with a confirmation message.

Considerations

- STATUS, entered with no range or condition parameters, reports the status of the last process TACL created, or for which TACL last paused, regardless of the CPU. If that process is no longer running, STATUS reports nothing.
- The STATUS command displays information for I/O processes when appropriate (for example, user 255,255 and the $OSP terminal). To eliminate the display of I/O processes, add PRI 199 to the command. For example:

```
STATUS *, USER, PRI 199
```

- A STATUS cpu,pin request displays both processes if the requested process is part of a process pair. If you request information about a system-image process, TACL displays the status of the single system-image process.
- If you specify a single process and the process does not exist, TACL returns “Process does not exist.”
- If the node name for a process cannot be retrieved by the STATUS program, the node name for the home terminal will be displayed.
- If you specify a range of processes and none of the processes exist, TACL displays nothing.

- File-name-template is not supported for the options PROG and LOADED used together. For example:

  ```
  STATUS *, PROG file-name-template, LOADED file-name-template
  ```

  will return no information.

- If file-name-template is specified with the LOADED option, STATUS will display all the processes associated with that loaded file.

- If the program file name for a process cannot be retrieved by the STATUS program, "`path to info down`" is displayed in the output instead of the file name.

  For the attenuated display:

  ```
  > STATUS *,TERM
  Process      Pri PFR %WT Userid  Program file        Hometerm

  $Y906  1,158 150     001 104,111 <Path to info down> \RCM.$Z1
  $MPMA  3,90  129     000 104,111 $A.XTK.XMSGX        $Z2
  $MP    5,80  150   R 000 104,111 $MG.TEST.TACL       $Z3
  ```

  For the detailed display:

  ```
  > STATUS $MP,DETAIL
  System: \PRUNE                     March 29, 2004  15:27
  Pid: 5,80    ($MP)    Primary
  READY
  Priority: 150
  Wait State: %000
  Userid: 104,111    (SDEV.FRED)
  Myterm: \PRUNE.$Z3
  Program File Name: <Path to info down>
  Swap File Name: $MG.#0012980
  Current Extended Swap File Name: $MG.TEST.TACLSEGF
  Process Time: 0:0:1.974
  Process Creation Time: March 29, 2004 15:22:24.101230
  Process States: LOGGED ON, FORCED LOW, RUNNABLE
  GMOMJOBID:
  ```

  If the processor for the disk process controlling the disk where the program file resides fails, the process file name is unavailable.

- If the output of the STATUS command includes the Swap File Name parameter and no swap file name has been set with either (or both) the SWAP *swap-file* option of the RUN command or the SET SWAP [$*volume-name*] command, a dummy file name, "$*vol*.#0", is returned. In this case, #*vol* is the name of the physical volume that the operating system has selected for storing the swap file. In the following example, "$SYSTEM.#0" is a dummy file name. The operating system is using the $SYSTEM volume if a swap file is required.

  ```
  Swap File Name: $SYSTEM.#0
  ```

- The output from the STATUS command can include information for OSS processes: the OSS program pathname, OSS arguments, and OSS process ID. Currently, only local OSS information can be obtained. For pre-D30 software RVUs, OSS information cannot be obtained.

  In the summary form of the STATUS command, the short form of the OSS program pathname is output. In the detailed form of the STATUS command, the fully qualified OSS program pathname, the OSS arguments, and the OSS process ID are output.

  The “Program file” display is limited to 26 characters. An OSS pathname can have a maximum of 1024 characters. If the fully qualified OSS program pathname is 26 or fewer characters, then the entire name is output, as illustrated in this example:

```
STATUS *,USER
Process         Pri PFR %WT Userid  Program file              Hometerm
$DCED X 0,339 155     004 255,161 /opt/dcelocal/bin/dced     $ZTNT.#P
```

  Otherwise, only the filename portion of the name is output.

  If the filename portion of the name is longer than 26 characters, then the filename is truncated to the first 23 characters and ellipses (...) are appended to the name, as illustrated in this example:

```
STATUS *,USER
Process         Pri PFR %WT Userid  Program file              Hometerm
$DCEE X 0,340 155     004 255,161 /opt/dcelocal/bin/abcsd... $ZTNT.#R
```

  If the file name portion of the name is not available, "`No OSS file name`" is output for the file name. If no path name is available, a generic ZYQ-file program file name is output.

  The OSS program pathname should not be used as input into a TACL command because it is not formatted in a way that a TACL process can process.

  In the detailed STATUS display, the OSS pathname, the first 1024 bytes of the arguments of the command that created the OSS process, and the OSS process ID (a unique identifier for an OSS process) are output:

```
STATUS 0,339,DETAIL
System: \FOXII                        November 13, 1995  10:53
Pid: 0,339    ($ZDCED)    Primary
Priority: 155
Wait State: %004    (LDONE)
Userid: 255,161    (SUPER.DCE)
Myterm: $ZTNT.#PTY000X
Program File Name: $OSS001.ZYQ00002.Z0000VK0
Swap File Name: $DCE.#0000249
Current Extended Swap File Name: $DCE.#0000250
Library File Name: $SYSTEM.ZSRL.LDCE
Process Time: 0:0:15.859
Process Creation Time: November 13, 1995 10:13:06.028548
Process States: RUNNABLE
GMOMJOBID:
OSS Pathname: /opt/dcelocal/bin/dced
```

```
OSS Arguments: -b
OSS PID: 882049029
```

- The output of the TACL STATUS command along with the DETAIL option now display ProgramDataModel, IPUAssociation, and IPUNumber depending on the following criteria:
  - ProgramDataModel is displayed for RVUs H06.24/J06.13 or later.
  - IPUAssociation is displayed for RVUs J06.16 or later.
  - IPUNumber is displayed for RVUs J06.03 or later.

Examples

1. To view the status of user 103, 141 enter:

```
14> STATUS *, user 103, 141

Process             Pri   PFR   %WT   Userid    Program file         Hometerm
$Y09M     B  0,193   168         001   103,141   $SYSTEM.SYS10.TACL   $ZTNT.#PTA8AE
9
          X  0,611   168         000   103,141   /bin.sh              $ZTNT.#PTA8AE
9
$Y09M        1,121   168     R   000   103,141   $SYSTEM.SYS10.TACL   $ZTNT.#PTA8AE
9
```

   This information is displayed:

   - The process name, if any.
   - The process type: If the process is the primary process, nothing is displayed. If the process is a backup process, “B” is displayed. If the process is an OSS process, “X” is displayed.
   - The CPU and process number.
   - The execution priority of the process.
   - PFR code: P indicates that the process contains privileged code; F indicates the process is waiting on a page fault; R indicates the process is on the ready list.
   - The wait state. This value is obtained from the wait field of the awake-wait word in the process control block (PCB) for the process. The wait field in the PCB can have these values:

```
wait-field.<8> wait on PON - CPU 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, a wait state of %003 means that bits 14 and 15 are set.
- The group-id, user-id of the process accessor.
- The name of the program file. For system processes, prog-name is $SYSTEM.SYSnn.OSIMAGE. (Subvolume $SYSTEM.SYSnn contains the operating system image currently in use; nn is a two-digit octal integer that identifies that subvolume.)
- The home terminal of the process.
- The name of the user library file, swap file, and extended swap file, if you requested the status of a single process (or a process pair) that is running with a user library (such as one specified with the LIB option of the RUN command) or swap files.

2. Including the DETAIL parameter in a STATUS command yields a display such as:

```
13> STATUS $TA8, DETAIL
System: \NEWYORK November 4, 2002 8:29
Pid: 1,31 ($TA8) Primary
Priority: 150
Wait State: %001 (LREQ)
Userid: 101,93 (SD.JKR)
Myterm: $TPQA8
Program File Name: $SYSTEM.SYS01.TACL
Swap File Name: $SYSTEM.#0000094
Current Extended Swap File Name: $TEMP.#0001203
Process Time: 0:1:47.870
Process Creation Time: October 22, 2002 22:12:19.599414
Process States: NO MESSAGES, LOGGED ON, RUNNABLE
GMOMJOBID:

System: \NEWYORK November 4, 2002 8:29
Pid: 2,35 ($TA8) Backup
Priority: 150
Wait State: %001 (LREQ)
Userid: 101,93 (SD.JKR)
Myterm: $TPQA8
Program File Name: $SYSTEM.SYS01.TACL
Swap File Name: $SYSTEM.#0000094
Current Extended Swap File Name: $TEMP.#0001203
Process Time: 0:0:2.238
Process Creation Time: October 22, 2002 22:12:21.426762
Process States: NO MESSAGES, LOGGED ON, RUNNABLE
GMOMJOBID:
```

The DETAIL option includes this information for the primary process and the backup process (if it exists):

- The system on which the process is running.

- The current system date and time.
- The CPU and process identification number of the process being displayed, the process name (if any), and, if named, whether this is the primary or backup process.
- The execution priority for the process.
- The wait-field value and the event that the process is waiting for. The values have this meaning:

%000	Process is running; or process was waiting on an event that has since occurred and is now ready to run, process is in call to DELAY, or process is suspended.
%001	Process is waiting for a message to occur on its $RECEIVE file.
%002	Process is waiting for a TMF subsystem request to finish, or user process is waiting for ENDTRANSACTION to finish.
%004	Process is waiting for input or output or interprocess request to finish.
%005	Process is waiting for call to AWAITIO for I/O completion on any file.

- User ID and name, Creator access ID (CAID) and its name, and Login name (ALIAS).

Note. User ID is equivalent to the Process access ID (PAID). CAID is displayed only if it differs from PAID. The Login name is displayed only if the creator of the process has logged in using the ALIAS name.

- The home terminal name of the process. If this terminal is connected to a remote system in a network, the node name precedes the terminal name, as in \CHICAGO.$TERM49.
- The program file name. For system processes, the program file name is $SYSTEM.SYSnn.OSIMAGE. This file contains the operating system image currently in use; num is a two-digit octal integer that identifies the subvolume.
- The swap file name (see RUN[D|V] Command on page 8-156).
- The extended swap file name, if it exists.
- The name of the user library file, if any. This line appears only when you request the status of a process or process pair that is running with a user library (such as one specified with the LIB option for the RUN command).
- The process time that has elapsed.
- Process creation time in local civil time format.

- The current process state, which can be one of these values:

```
[IN SYSTEM MAB] [,NO MESSAGES]
[,TEMPORARY] [,LOGGED ON] [,PENDING]
STARTING
RUNNABLE
SUSPENDED
CPU BOUND
DEBUG MAB
DEBUG BREAKPOINT
DEBUG TRAP
DEBUG REQUEST
FORCED LOW
INSPECT MAB
INSPECT BREAKPOINT
INSPECT TRAP
INSPECT
REQUEST
SAVE ABEND
TERMINATING
TSN LOGON
```

- The job ancestor of the process.

3. To stop all of your processes on the terminal where your TACL is running (except the TACL process itself):

```
STATUS *, TERM, STOP
```

The following confirmation message is displayed:

```
This command will display the processes satisfying the
command parameters and then stop them. Do you really want to
stop the processes (y/[n])?
```

A process, started by an alias (sspaul) from a progid-ed (super.super) object owned by support.prs, will have the following as the output to the STATUS command with the DETAIL option.

```
Pid: 2,110    ($PIPO)    Primary
Priority: 148
Wait State: %004    (LDONE)
Process access id: 20,33    (SUPPORT.PRS)
Creator access id: 255,255    (SUPER.SUPER)
Login name: sspaul
Myterm: $ZN018.#PT05WPE
Program File Name: $SYSTEM.PJ00WTAL.VPROC
Swap File Name: $SYSTEM.#0
Current Extended Swap File Name: $SYSTEM.#0
Process Time: 0:0:0.004
Process Creation Time: April 2, 2009 20:02:36.642570
Process States: RUNNABLE
GMOMJOBID:
```

STOP Command

Use the STOP command to request termination of a running process.

```
STOP [ [\node-name.]{$process-name | cpu,pin} ]
```

\node-name

is the system where the process resides.

$process-name

is the name of the process or process pair.

cpu,pin

is the CPU number and process number for the process.

Considerations

If a process terminates successfully, TACL does not display a message. Otherwise, the STOP command displays information about the outcome of the termination request. The possible outcomes are listed in Table 8-9.

Table 8-9. STOP Command Messages

Message	Meaning
Non-existent *process-name*	The process has already stopped.
STOP message has been queued.	The process was not stopped, but the operating system queued the request.
Could not stop *process-name*; insufficient privilege.	The process could not be stopped because the process issuing the STOP command did not have sufficient privilege.
Could not stop *process-name*; stop error *error-number*.	The process could not be stopped, and TACL does not recognize the reason for the failure.

- If the process cannot be terminated immediately, the STOP operating system procedure queues the request.
- If you do not specify a process (*cpu,pin* or *$process-name*), STOP stops the process most recently started by TACL or the one for which TACL most recently paused, if that process is still running.
- The super ID can stop any user process. A group manager can stop any process whose creator accessor ID matches any user ID in the group. Other users can stop processes that have creator accessor IDs that match their user IDs or those that have their stop mode set to zero. (See the *Guardian Procedure Calls Reference Manual* for remote process restrictions.)

- STOP cannot stop the current TACL or its backup. Use the built-in function #STOP instead.

Examples

1. To stop the last process you started from the current TACL, enter:

```
14> STOP
15>
```

2. If you started the process whose *cpu,pin* is 0,18, or if you are the super ID, you can stop the process by entering:

```
14> STOP 0,18
15>
```

3. If you are a group manager and a member of your group started the process named $END, or if you are the super ID, you can stop the process $END by entering the process name or *cpu,pin*:

```
14> STOP $END 15>
```

SUSPEND Command

Use the SUSPEND command to temporarily suspend a process (or process pair) to prevent it from competing for system resources.

```
SUSPEND [ [\node-name.]{$process-name | cpu,pin } ]
```

\node-name

is the system where the process resides.

$process-name

is the name of the process or process pair.

cpu,pin

is the CPU number and process number for the process.

Considerations

- A suspended process or process pair cannot execute instructions until you reactivate it. To reactivate a suspended process or process pair, use the ACTIVATE command. (Also, another process can call the ACTIVATEPROCESS procedure to reactivate the suspended process or process pair.)
- If you do not specify a process (*cpu,pin* or $*process-name*), SUSPEND suspends the process most recently started by the current TACL or the one for which TACL most recently paused, if that process is still running.
- TACL does not suspend the process (or process pair) you name in your SUSPEND command until that process is ready to execute instructions. Therefore, any outstanding waits (such as for I/O completion) are satisfied before the process or process pair is suspended.
- Standard users can suspend only processes that they started (and descendants of those processes). That is, the current user's process accessor ID must match the process accessor ID of the process to be suspended. (See the *Expand Network Management and Troubleshooting Guide* for restrictions on remote processes.)
- A group manager can suspend any process whose process accessor ID matches any user ID in the group.
- The super ID can suspend any process.
- SUSPEND cannot suspend the current TACL or its backup.

Examples

1. To suspend the last process you started from the current TACL, enter:

```
14> SUSPEND
15>
```

2. If you started a process with cpu,pin 0,18, or if you are the super ID, you can suspend the process by entering:

```
14> SUSPEND 0,18
15>
```

3. If you are a group manager and a member of your group started the process whose name is $CLOCK, or if you are the super ID, you can suspend the process by entering the process name or cpu,pin:

```
15> SUSPEND $CLOCK
16>
```

SWITCH Command

Use the SWITCH command to make your TACL backup the primary process, initializing itself as though you had just logged on to it. The former primary process becomes the backup. The SWITCH command is an alias for the #SWITCH built-in function.

```
SWITCH
```

Considerations

- To use the switch command, your TACL must have a backup process. To create a backup process, include it in the TACL RUN command or use the BACKUPCPU command.
- The SWITCH command must be entered interactively. Do not include the SWITCH command in an IN file specified in a command to run TACL; if you do so, TACL performs the switch before processing other commands-and each switch causes an initialization so that the TACL process continues to switch processors.
- SWITCH establishes an initial logon state, resetting all ASSIGNs, PARAMs, and DEFINEs, invoking the TACLLOCL file and your TACLCSTM file, and setting the history buffer index to 1.
- If an error occurs while TACL is trying to create the backup process or if the backup CPU is down, TACL waits 3 minutes before trying to create the backup.
- All events, such as a backup-create error or an I/O error event, and the error details are logged to the primary or $0 collector. This format is used:

```
TACL BACKUP CREATE ERROR: error, DETAIL: error-detail
```

error

is the error returned by PROCESS_CREATE.

error-detail

is the error detail value returned by PROCESS_CREATE.

For more information on the $0 collector, see the *EMS Manual*.

Example

Assume that the primary TACL process that controls your terminal is running in CPU 5, and the backup TACL process is running in CPU 4. (To display the CPU numbers of the processors where your TACL processes are running, use the WHO or STATUS command.)

You can switch the functions of these processes (make the TACL process running in CPU 4 the primary process and the process running in CPU 5 the backup) by entering the SWITCH command:

```
24> SWITCH
```

SYSTEM Command

Use the SYSTEM command to set the default system until you change it again or log off. This command applies only to systems that are available in a network.

```
SYSTEM [ \node-name ]
```

\node-name

is the name of a new default system. If you omit \ node-name, the system on which your TACL is running becomes the current default.

Considerations

- The system you specify in a SYSTEM command is temporarily in effect. After you enter a LOGON command, or a SYSTEM or VOLUME command with no following parameters, your default system is again your logon default. (To change your logon defaults, use the DEFAULT program.)
- If you are running a remote TACL process, entering SYSTEM with no following parameters establishes that remote system as your default system, instead of the local system to which your terminal is connected.
- These commands are not equivalent:

  ```
  14> SYSTEM \ local-node-name
  ```

  ```
  14> SYSTEM
  ```

  The first invocation causes the network restrictions on file-name lengths to take effect; the second does not. See the *Expand Network Management and Troubleshooting Guide* for information on network file-name restrictions.
- To view the default system, use the WHO command or the #DEFAULTS built-in variable. If the WHO command does not display a current system, your current system is the local system.

Examples

1. To specify \LONDON as the current default system, enter:

   ```
   14> SYSTEM \LONDON
   ```

   When you enter the WHO command, the display includes the current default system (omitted when it is the same as the saved default):

   ```
   15> WHO
   ...
   Current volume: $BOOKS.TACL Current system: \LONDON
   ```
2. To return to your saved default system (established by the DEFAULT command):

   ```
   16> SYSTEM
   ```

SYSTIMES Command

Use the SYSTIMES command to display the current date and time (in local civil time and Greenwich mean time), the date and time when the system was last cold loaded, and the date and time SYSGEN was last run.

```
SYSTIMES
```

Considerations

- The SYSTIMES command displays four lines of information giving you the date and time as shown here:

```
ddmmmyyyy, hh:mm:ss.mmmuuu LCT
ddmmmyyyy, hh:mm:ss.mmmuuu GMT
ddmmmyyyy, hh:mm:ss.mmmuuu Cold Load (LCT)
ddmmmyyyy, hh:mm:ss.mmmuuu SYSGEN (LCT)
```

dd	The day of the month (01, 02, ... , 31)
mmm	The three-letter abbreviation for the month of the year (JAN, FEB, ... , DEC)
yyyy	The 4-digit calendar year from 1975 through 9999
hh	The hour of the day (00, 01, ... , 23)
mm	The minutes of the hour (00, 01, ... , 59)
ss	The seconds of the minutes (00, 01, ... , 59)
mmmuuu	The millisecond and microsecond of the second (000000, 000001, ... , 999999)
LCT	Indicates that the date and time are shown in local civil time
GMT	Indicates that the date and time are shown in Greenwich mean time
`Cold Load (LCT)`	Indicates the date and time of the most recent cold load This time is given in local civil time.
`SYSGEN (LCT)`	Indicates the date and time that SYSGEN was run to produce the current system image This time is given in local civil time.

- The SYSTIMES command applies only to the local system. To obtain the times from a remote system, you must first start a TACL process there and then enter the SYSTIMES command.

Example

To display the various system times, enter:

```
37> SYSTIMES
6 Aug 1992, 12:09:54.589512 LCT
6 Aug 1992, 20:09:54.589512 GMT
30 Jul 1992, 19:18:03.750427 Cold Load (LCT)
9 Jul 1992, 11:52:13.090000 SYSGEN (LCT)
```

TACL Program

Enter the TACL program name to start a TACL process on your local system or on a remote system (if your system is part of a network).

```
[\node-name.]TACL [ / run-option [ , run-option ] ... / ]
  [ backup-cpu-num ] [ ; parameter [ , parameter ] ]
```

\node-name

is the name of the system on which TACL is to run. This parameter is valid only for systems that have a node name (those that are part of a network). If you omit \ node-name, the TACL process runs on the system from which you issued the TACL command.

run-option

is any of the options described in the RUN[D|V] Command on page 8-156.

backup-cpu-num

specifies the processor where the backup for the new TACL process is to run. Specify *backup-cpu-num* as an integer in the range from 0 through 15. If you omit this parameter, no backup process is created. You can specify a backup process only if the TACL process is a named process (see the NAME option of the RUN[D|V] Command on page 8-156).

parameter

is an operating parameter for the TACL process. It can be one of these:

```
ABENDONABEND
HOMETERM
PORTTACL
SEGVOL $volume-name
STOPONABEND
```

ABENDONABEND

specifies that this TACL process writes the message: "`TACL stopped by a process ABEND/STOP`" to the current OUT file and terminates abnormally (ABENDs) with an abend completion code when a child process (a process started by this TACL process) terminates with no completion code or with an abnormal completion code.

HOMETERM

specifies that TACL is to use the device specified by the TERM run-option as the home terminal. If you omit HOMETERM, if the TACL IN file is the same as the TACL OUT file, and if TACL is not in server mode, TACL uses its IN file device as the home terminal, regardless of any specification by the TERM

option. If the IN file is the same as the OUT file and the TACL process is not named, TACL does not set its home terminal.

A process started by TACL inherits its home terminal unless the RUN command that initiates the process specifies a different home terminal.

PORTTACL

specifies that the TACL being started is a port TACL (for example, a modem port for a dial-in line or an X25 connection).

When using this parameter, the #SETCONFIGURATION option STOPONFEMODEMERR should be ON. The default setting for the #SETCONFIGURATION option STOPONFEMODEMERR is OFF.

The relationship between PORTTACL and STOPONFEMODEMERR is summarized in this table.

PORTTACL	STOPONFEMODEMERR	Outcome
Specified	ON	When error 140 (FEMODEMERR) is encountered on its input, this TACL process issues a modem disconnect message, goes into the logged-off state, and waits for a modem connection message. When this message is received, a user can log on. The process does not have to be restarted.
Specified	OFF	When error 140 (FEMODEMERR) is encountered on its input, this TACL process ignores it and continues in whatever the current state is at the time.
Absent	ON	When error 140 (FEMODEMERR) is encountered on its input, this TACL process issues a modem disconnect message, goes into the logged-off state, and stops.
Absent	OFF	When error 140 (FEMODEMERR) is encountered on its input, this TACL process ignores it and continues in whatever the current state is at the time.

- To start a TACL with the PORTTACL parameter, you must have a group ID of 255, regardless of the setting of the TACL configuration parameter STOPONFEMODEMERR. If your group ID is not 255, the TACL process terminates abnormally (ABENDs). It is the responsibility of the super-group

user to decide which TACL processes need to be started with the PORTTACL startup parameter.

- If a backup CPU was specified for the TACL process being started with the PORTTACL startup option and the primary TACL fails, the backup TACL process inherits the PORTTACL setting from the primary TACL process.
- The PORTTACL startup parameter is only valid as a TACL process startup parameter option. It is not a valid LOGON parameter option.
- If a TACL process is not the process controlling modem port access, it does not receive disconnect or error messages from the port modem, including error 140 (STOPONFEMODEMERR). It is the responsibility of the process that controls port access and receives messages from the port modem to process appropriately disconnect and error messages, including message140.
- If both TACL and $CMON processes are configured for modem port connections and the STOPONFEERROR option is enabled, ensure that the $CMON program can appropriately process the PORTTACL parameter for TACL process startup.

△ **Caution.** If a port is accessed only by means of a TACL process (that is, there is no Safeguard or other type of port access control) a potential system security breach exists unless the PORTTACL option is specified and STOPONFEMODEMERR is set to ON.

In the case where the TACL process stops, a user can then start a TACL process, under the user's control. In the cases where the TACL process continues to run in the logged-on state, and the connection is resumed, any user can access the TACL process without logging on and assume the identity of the original user.

SEGVOL $*volume-name*

specifies the name of a volume to be used by your extended segment, which holds the TACL process variables. The default is the volume that contains the default subvolume of the user who is logging on to the TACL.

STOPONABEND

specifies that this TACL process sends the message "TACL stopped by a process ABEND/STOP" to the current OUT file and terminates normally with a normal completion code when a subordinate process (a process started by this TACL process) terminates with no completion code or with an abnormal completion code.

Considerations

- If you start a TACL process with the STOPONABEND option and specify a backup CPU for the TACL process, and if the CPU fails, the new primary TACL process initializes itself as if you had just logged on to it. The new process inherits the STOPONABEND setting and reprocesses the IN file.

- To run TACL as a server process, set the IN file to $RECEIVE. For more information, see the *TACL Programming Guide*.
- If the IN file is the same as the OUT file and the TACL process is not named, TACL does not set its home terminal.
- When using the ABENDONABEND and STOPONABEND parameters:

  If a TACL process is configured with either ABENDONABEND or STOPONABEND:

  - If a child process is started in a different processor and that processor fails, the TACL process assumes the subordinate process has terminated abnormally and executes accordingly.
  - If a child process is started as a NonStop process pair, both members of the pair must terminate abnormally, or the processors where these child processes are running fail before the TACL process assumes the child process pair has terminated abnormally and executes accordingly.
  - If a child process or process pair is started with the NOWAIT option and terminates, the TACL process displays any error message it may have received from the child process but continues to execute.

  If a TACL process is configured with the STOPONABEND parameter and starts a child process and this child process is stopped by another process:

  - If the JOBID of the of the stopping process is the same as that of the stopped process, the TACL process executes accordingly. It stops.
  - If the JOBID of the of the stopping process is different from that of the stopped process or 0, the TACL process does not execute accordingly. Instead, it terminates abnormally (ABENDs).
  - If a TACL process is started as a process pair (by specifying a BACKUP CPU as a startup parameter) and the primary process fails, when the backup process takes over it inherits the STOPONABEND or ABENDONABEND setting. This takeover process reprocesses the IN file, and begins execution in the logged-on state. If. however, the TACL process is the cold load TACL, then it begins execution in the logged-off state.
  - If a TACL process is started as a system load process pair and also controls a modem port and the primary process fails, when the backup process takes over, it inherits the STOPONABEND or ABENDONABEND setting. This takeover process reprocesses the IN file and begins execution in the logged-off state.

  The STOPONABEND or ABENDONABEND parameter specified at TACL startup is the default setting for all TACL logon sessions started from that TACL. You can override the default setting by specifying the other parameter in the LOGON

command. In this case, ABENDONABEND overrides the default setting (STOPONABEND) just for this logon session.

```
12> TACL /NAME/ ;STOPONABEND
TACL 1> LOGON SOFTWARE.JANE ;ABENDONABEND
Password:
```

If you start a TACL process with both the STOPONABEND and ABENDONABEND parameters, the last parameter specified in the list overrides the first one. In this example, STOPONABEND overrides ABENDONABEND.

```
12> TACL /NAME/ ;ABENDONABEND
TACL 1> LOGON SOFTWARE.JANE ;STOPONABEND
Password:
```

The completion code returned by the TACL process when it generates the "*ERROR* TACL stopped by a process ABEND/STOP - PID:" message is specified in the :_COMPLETION (C-series systems) or :_COMPLETION^PROCDEATH (D-series systems) variable:

```
*ABEND*
*ERROR* TACL stopped by a process ABEND/STOP - PID:
ABENDED: $MP
```

```
29> OUTVAR :_COMPLETION
_COMPLETION(0)
  MESSAGECODE(0:0)
                -6
  PROCESS(0:0)    $MP
  HEADERSIZE(0:0) 0
  CPUTIME(0:0)    1854774
  JOBID(0:0)      0
  COMPLETIONCODE(0:0)
                5
  INTERNAL(0)
    TERMINATIONINFO(0:0)
                0
    SUBSYSTEM(0:0)
  TEXTLENGTH(0:0) 0
  TEXT(0:79)
```

```
30> OUTVAR :_COMPLETION^PROCDEATH
_COMPLETION^PROCDEATH(0)
  Z^MSGNUMBER(0:0)
-101
  Z^PHANDLE(0:0)  512.675.3.152.0.0.574.34458.0.175
  Z^CPUTIME(0:0)  1854774
  Z^JOBID(0:0)    0
  Z^COMPLETION^CODE(0:0)
                  5
  Z^TERMINATION^CODE(0:0)
                  0
  Z^SUBSYSTEM(0:0)

  Z^KILLER(0:0)   65535.65535.65535.65535.65535...
  Z^TERMTEXT^LEN(0:0)
```

```
                0
Z^PROCNAME(0)
  ZOFFSET(0:0)    82
  ZLEN(0:0)       20
Z^FLAGS(0:0)    1
Z^RESERVED(0:2) 1 1 0
Z^DATA(0)
  BYTE(0:111)     \PRUNE.$MP:150608489
```

Examples

1. This example shows how to start an interactive TACL process from an existing TACL process:

   ```
   13> TACL / IN $term1, OUT $term1, CPU 2, PRI 150, NOWAIT,&
   13> &NAME $C106 / 3
   ```

   After you enter this command:

   - A new TACL process starts; it accepts commands from, and displays its output on, terminal $TERM1.
   - The name of the new TACL process is $C106.
   - The primary TACL process for $TERM1 is running in processor 2. The backup process is running in processor 3.
   - The execution priority of the new process is 150.
   - Because the NOWAIT option was used, you can immediately execute another command from the current TACL without waiting for the new TACL process to terminate.

2. This example shows how to start a TACL process that uses a command file for input:

   ```
   14> TACL / IN comfile, OUT listing, NOWAIT /
   ```

   After you enter this command:

   - A new TACL process executes the commands contained in the file COMFILE.
   - Output from the new TACL process is sent to the file LISTING.
   - The NOWAIT option means that control of the terminal returns to the original TACL process while the new TACL process runs. You can now enter new commands.

TIME Command

Use the TIME command to display the current setting of the system date and time-of-day clock in the format:

mmm dd, yyyy, hh:mm:ss

```
TIME
```

Considerations

- The year is the 4-digit calendar year, from 1975 through 9999.
- You can execute a TIME command without having logged on (with the LOGON command) to a system.

Examples

If you are logged on, the TIME command displays:

```
88> TIME
July 9, 2002 15:57:39
```

If you are logged off, the TIME command uses a slightly different format:

```
90> TIME
09 JULY 2002, 15:59
```

USE Command

The USE command defines the list of directories (in the built-in variable #USELIST) that your TACL searches to find existing variables if they are not in your home directory.

```
USE [ directory-name [ [,] directory-name ] ... ]
```

directory-name

is the name of an existing variable level of type DIRECTORY. If omitted, the use list is set to:

```
:, :UTILS, :UTILS:TACL
```

Considerations

- When you invoke a name, TACL first searches the home directory for a variable of that name, and then in #USELIST. If these searches are unsuccessful, TACL searches #PMSEARCHLIST for a macro or program file of that name.
- The USE command ensures that your use list always includes the directories : (the root), :UTILS, and :UTILS:TACL. If you omit any of these directories in your arguments to USE, TACL automatically appends them to the use list in the order shown. Access to, and operation of, TACL commands depends on their presence.
- Directories are put into #USELIST in the order in which they are specified.
- The list set by this command can be displayed with the ENV command.
- You can save and restore the use list by pushing and popping #USELIST.
- If you detach a segment contained in the current use list or a pushed use list, the directory is removed from the use list.
- The use list can contain up to 100 directories.

Example

```
13> USE MYDIR
14> ENV
Home :MYDIR.1
Pmsearch $MYVOL.MINE, $SYSTEM.SYSTEM
System \MYSYS
Use :MYDIR.1, :, :UTILS.1, :UTILS.1:TACL.1
Volume \MYSYS.$WORK.PROJECT, "NUNU"
```

USERS Program

Use the USERS program to list user attributes for a particular user or range of users.

```
USERS [ / run-option [ , run-option ] ... / ] [ range ]
```

run-option

is any of the options described in the RUN[D|V] Command on page 156.

range

specifies a particular user or group of users to be listed. Note that range refers to the local system only. The allowable range specifications and their meanings are:

(blank)

lists current user only.

group-id,user-id

lists user with specified user number.

*group-id,**

lists all users in specified group.

group-name.user-name

lists named user only.

*[group-name.]**

lists all users in named group. If you omit *group-name,* USERS, lists users in your group.

, or *.*

lists all users.

Examples

1. To display the USERS listing for yourself (the currently logged-on user), enter:

   ```
   13> USERS
   ```

2. To find out the user name associated with the user ID 8,44, enter:

   ```
   14> USERS 8,44
   ```

   The USERS program displays information such as this:

   ```
   GROUP USER I.D. # SECURITY DEFAULT VOLUMEID
   MANUF .FRED 008,044 NUNU $BIG.BAD
   ```

3. You can get this information for all users in the PARTS group by entering:

```
15> USERS PARTS.*

GROUP USER I.D. # SECURITY DEFAULT VOLUMEID

PARTS .CLYDE 001,000 GGGO $SYSTEM.ENGINE
PARTS .JOE 001,001 CUCU $SYSTEM.TRANS
PARTS .MARY 001,002 AOGO $SYSTEM.WHEELS
PARTS .MANAGER 001,255 GGGA $SYSTEM.PAYROLL
```

VARIABLES Command

Use the VARIABLES command to display the names of all variables in a directory.

```
VARIABLES [ directory-name ]
```

directory-name

is the name of an existing variable level of type DIRECTORY.

Considerations

- If you omit directory-name, the home directory is assumed.
- If the directory being displayed contains a directory, that inner directory name is flagged with an asterisk (*). The names of variables in that directory are not listed.

Example

This example lists variables in the home directory:

```
39> VARIABLES

Directory :

*MYSEG NEWPTIME OLDPTIME PROCESSID
*UTILS _PROMPTER _TACLBASE_FILE
```

VARINFO Command

Use the VARINFO command to display attribute information about one or more of your variables.

```
VARINFO [ variable [ [,] variable ] ... ]
```

variable

is the name of an existing variable.

Considerations

- If you omit variable, VARINFO displays information about all variables in your home directory.
- VARINFO displays information similar to this:

```
Variable L/D Type Frm Mode File Process
MYDIR 1/1 DIRECTORY 0 SHARED $V1.SV2.FILE3
```

Variable

indicates the name of the variable.

L/D L

indicates the specific level of a variable about which you want information; D (depth) indicates the total number of levels. Type indicates the type of variable (directory, text, and so on).

Frm

indicates the frame number in which the variable level being examined was created.

Mode

indicates different types of status of a variable depending on whether it is associated with #REQUESTER or #SERVER, or with a segment:

- Mode is one of these options for #REQUESTER:

  ```
  READ WRITE
  ```

- Mode is one of these options for #SERVER:

  ```
  IN IN_DYNAMIC OUT STATUS PROMPT
  ```

- Mode is one of these options for a segment:

```
PRIVATE SHARED
```

File

is the name of #SERVER or the file opened by #REQUESTER. In the case of a segment, it is the name of the segment file with which the variable is associated.

Process

is the process associated with an implicit #SERVER; that is, the process that started with INV, OUTV, or STATUS specifying the variable.

- For an unnamed process, a process ID (that is, *cpu,pin)* is returned in the Process field.
- For a named process, the process name is returned in the Process field.
- If the process name is not available, nothing is returned in the Process field.

VARTOFILE Command

Use the VARTOFILE command to copy the data in a variable to a file.

```
VARTOFILE variable-level file-name
```

variable-level

is the name of an existing variable level from which data is to be copied. It must not be a DIRECTORY, a STRUCT, or a STRUCT item.

file-name

is the name of the file that is to receive the copy. If the file exists, it must be of a type that can be written to by the sequential I/O (SIO) facility. The new data is appended to the existing data. If the file does not exist, TACL creates an edit-format file.

Considerations

- Lines longer than 239 characters are truncated to 239 characters.
- For security, TACL makes a temporary copy of the variable while this command is being executed.
- I/O is done in PLAIN format. This means that the internal representations of the metacharacters [, |, and] in TACL statements are not translated to their external representations when written to the file.

VCHANGE Command

Use the VCHANGE command to change all occurrences of one string to another string in a range of lines within a variable.

```
VCHANGE [ / option [ , option ] ... / ] variable-level
  string-1 string-2 [ range ]
```

option

is any of these:

```
OUT file-name
QUIET
TO variable-level
```

OUT *file-name*

specifies a device, or a sequential file accessible to the sequential I/O (SIO) facility, that is to receive the changed lines. The listing includes line numbers. If you omit this option, TACL writes the listing to its current OUT file. If the QUIET option is present, this option is ignored.

If you specify an OUT file that does not exist, TACL creates an EDIT file named list-file. If you specify an OUT file that already exists, TACL appends the information to the end of the file.

QUIET

suppresses listing of changed lines, even if the OUT option is supplied.

TO *variable-level*

is the name of an existing variable level to which a listing of all changed lines is to be appended. Line numbers are not included. The QUIET option has no affect on this option.

variable-level

is the name of an existing variable level containing the lines to be changed. It must not be in a shared segment, and must not be a DIRECTORY, a STRUCT, or a STRUCT item.

string-1

is the string to be changed wherever it occurs in the specified range of lines. A string is the name of a variable level, text enclosed in quotation marks, or a concatenation of such entities. The concatenation operator is '+' (the apostrophes are required).

string-2

is the string that is to replace *string-1*.

range

specifies the line or lines in which the change is to occur. If you omit it, all lines are included. A range specification can be any of these:

```
A
line-num
line-num / line-num
```

A

specifies all lines in the variable level.

line-num

specifies an individual line number. It can be any of these:

```
F
L
number
```

F

specifies the first line in the variable level.

L

specifies the last line in the variable level.

number

is a positive integer identifying a specific line.

Considerations

- VCHANGE is not case-sensitive. If, for example, you specify:

  ```
  VCHANGE var "The" "A"
  ```

  VCHANGE changes all occurrences of The, the, THE, or any other combinations of uppercase or lowercase T, H, and E. For case-sensitive operations, you can use the string-handling functions.

- If *string-1* is empty, no change occurs.

- If *string-1* or *string-2* contains TACL metacharacters, the setting of the #INFORMAT built-in variable (for input to the IN file, including input to an interactive TACL process) or the ?FORMAT Directive on page 5-6 (for text in TACL programs) can affect how TACL interprets the string. For more information, see the

#INFORMAT Built-In Variable on page 9-196 or the ?FORMAT Directive on page 5-6.

Example

If these variables have the contents shown:

Var	Listvar
THE QUICK BROWN	12345678910
FOX JUMPED OVER	11121314151617
THE LAZY DOG	18192021
TWICE A DAY	
EXCEPT TUESDAYS	

then the command:

```
VCHANGE /TO listvar/ var "THE" "A" 1/4
```

causes them to contain:

Var	Listvar
A QUICK BROWN	12345678910
FOX JUMPED OVER	11121314151617
A LAZY DOG	181920217
TWICE A DAY	A QUICK BROWN1
EXCEPT TUESDAYS	A LAZY DOG

and causes this to be written to the TACL OUT file:

```
1 A QUICK BROWN
3 A LAZY DOG
```

VCOPY Command

Use the VCOPY command to copy a range of lines from one variable and insert them at a given line position in another variable.

```
VCOPY [ / option [ , option ] ... / ] source-var range
  dest-var dest-line
```

option

is any of these:

```
OUT file-name
QUIET
TO variable-level
```

OUT *file-name*

specifies a device, or a sequential file accessible to the sequential I/O (SIO) facility, that is to receive the copied lines. The listing includes line numbers. If you omit this option, TACL writes the listing to its current OUT file. If the QUIET option is present, this option is ignored.

If you specify an OUT file that does not exist, TACL creates an EDIT file named list-file. If you specify an OUT file that already exists, TACL appends the information to the end of the file.

QUIET

suppresses listing of copied lines even if the OUT option is supplied.

TO *variable-level*

is the name of an existing variable level to which a listing of all copied lines is to be appended. Line numbers are not included. The QUIET option has no affect on this option.

source-var

is an existing variable level containing the lines to be copied. It must not be in a shared segment, and must not be a DIRECTORY, a STRUCT, or a STRUCT item.

range

specifies the line or lines to be copied. A range specification can be any of these:

```
A
line-num
line-num / line-num
```

A

specifies all lines in the variable level.

line-num

specifies an individual line number. It can be any of these:

```
F
L
number
```

F

specifies the first line in the variable level.

L

specifies the last line in the variable level.

number

is an integer identifying a specific line.

dest-var

is an existing variable level that is to receive the copy. It must not be in a shared segment, and must not be a DIRECTORY, a STRUCT, or a STRUCT item.

dest-line

specifies the line number in the destination variable level at which the copied lines are to be inserted. This entry can be any of these:

```
F
L
number
```

F

specifies the first line in the variable level.

L

specifies a new line past the last line in the variable level.

number

is an integer identifying a specific line. It must be greater than zero and not greater than the number of lines in the variable level, plus one (it can specify a new line past the last existing line).

Considerations

- *source-var* and *dest-var* must not be identical.
- Copied lines are inserted immediately preceding *dest-line*.

Example

If these variables have the contents shown:

Srcvar	Dstvar	Listvar
THE QUICK BROWN	ABCDEFG	12345678910
FOX JUMPED OVER	HIJKLMNOPQRST	1121314151617
THE LAZY DOG	UVWXYZ	18192021
TWICE A DAY		
EXCEPT TUESDAYS		

the command:

```
VCOPY /TO listvar/ srcvar 2/4 dstvar 3
```

causes them to contain:

Srcvar	Dstvar	Listvar
THE QUICK BROWN	ABCDEFG	12345678910
FOX JUMPED OVER	HIJKLMNOPQRST	11121314151617
THE LAZY DOG	FOX JUMPED OVER	18192021
TWICE A DAY	THE LAZY DOG	FOX JUMPED OVER
EXCEPT TUESDAYS	TWICE A DAY	THE LAZY DOG
	UVWXYZ	TWICE A DAY

and causes this to be written to the TACL OUT file:

```
3 FOX JUMPED OVER
4 THE LAZY DOG
5 TWICE A DAY
```

VDELETE Command

Use the VDELETE command to delete a range of lines from a variable.

```
VDELETE [ / option [ , option ] / ... ] variable-level range
```

option

is any of these:

```
OUT file-name
QUIET
TO variable-level
```

`OUT` *file-name*

specifies a device, or a sequential file accessible to the sequential I/O (SIO) facility, that is to receive the deleted lines. The listing includes line numbers. If you omit this option, TACL writes the listing to its current OUT file. If the QUIET option is present, this option is ignored.

If you specify an OUT file that does not exist, TACL creates an EDIT file named list-file. If you specify an OUT file that already exists, TACL appends the information to the end of the file.

`QUIET`

suppresses listing of deleted lines even if the OUT option is supplied.

TO variable-level

is the name of an existing variable level to which a listing of all deleted lines is to be appended. Line numbers are not included. The QUIET option has no affect on this option.

variable-level

is an existing variable level from which lines are to be deleted. It must not be in a shared segment, and must not be a DIRECTORY, a STRUCT, or a STRUCT item.

range

specifies the line or lines to be deleted. A range specification can be any of these:

```
A
line-num
line-num / line-num
```

A

specifies all lines in the variable level.

line-num

specifies an individual line number. It can be any of these:

F
L
number

F

specifies the first line in the variable level.

L

specifies the last line in the variable level.

number

is an integer identifying a specific line.

Example

If these variables have the contents shown:

Var	Listvar
THE QUICK BROWN	12345678910
FOX JUMPED OVER	11121314151617
THE LAZY DOG	18192021
TWICE A DAY	
EXCEPT TUESDAYS	

the command:

```
VDELETE /TO listvar/ var 2/4
```

causes them to contain:

Var	Listvar
THE QUICK BROWN	12345678910
EXCEPT TUESDAYS	11121314151617
	18192021
	FOX JUMPED OVER
	THE LAZY DOG
	TWICE A DAY

and causes this to be written to the TACL OUT file:

```
2 FOX JUMPED OVER
3 THE LAZY DOG
4 TWICE A DAY
```

VFIND Command

Use the VFIND command to find all lines containing occurrences of a specified string in a range of lines within a variable.

```
VFIND [ / option [ , option ] / ... ] variable-level string
  [ range ]
```

option

is any of these:

```
OUT file-name
QUIET
TO variable-level
```

OUT *file-name*

specifies a device, or a sequential file accessible to the sequential I/O (SIO) facility, that is to receive the listing of lines in which string is found. The listing includes line numbers. If you omit this option, TACL writes the listing to its current OUT file. If the QUIET option is present, this option is ignored. If you specify an OUT file that does not exist, TACL creates an EDIT file named list-file. If you specify an OUT file that already exists, TACL appends the information to the end of the file.

QUIET

suppresses listing of lines in which string appeared even if the OUT option is supplied.

TO *variable-level*

is the name of an existing variable level to which a listing is to be appended of all lines in which string is found. Line numbers are not included. The QUIET option has no affect on this option.

variable-level

is the name of an existing variable level in which the search is to be made. It must not be in a shared segment, and must not be a DIRECTORY, a STRUCT, or a STRUCT item.

string

is the string to be found. A string is the name of a variable level, text enclosed in quotation marks, or a concatenation of such entities. The concatenation operator is '+' (the apostrophes are required).

range

specifies the line or lines in which the search is to be made. If you omit it, TACL searches all lines. A range specification can be any of these:

```
A
line-num
line-num / line-num
```

A

specifies all lines in the variable level.

line-num

specifies an individual line number. It can be any of these:

```
F
L
number
```

F

specifies the first line in the variable level.

L

specifies the last line in the variable level.

number

is a positive integer identifying a specific line.

Considerations

- VFIND is not case-sensitive. If, for example, you specify

  ```
  VFIND var "The"
  ```

  VFIND locates the first occurrence of The, the, THE, or any other combination of uppercase or lowercase T, H, and E. For case-sensitive operations, you can use the string-handling functions.

- If string contains TACL metacharacters, the setting of the #INFORMAT built-in variable (for input to the IN file, including input to an interactive TACL process) or the ?FORMAT Directive on page 5-6 (for text in TACL prog the #INFORMAT Built-In Variable on page 9-196 or the ?FORMAT directive.

Example

If these variables have the contents shown:

Var	Listvar
THE QUICK BROWN	12345678910
FOX JUMPED OVER	11121314151617
THE LAZY DOG	18192021
TWICE A DAY	
EXCEPT TUESDAYS	

the command:

```
VFIND /TO listvar/ var "THE" 1/4
```

causes them to contain:

Var	Listvar
THE QUICK BROWN	12345678910
FOX JUMPED OVER	11121314151617
THE LAZY DOG	18192021
TWICE A DAY	THE QUICK BROWN
EXCEPT TUESDAYS	THE LAZY DOG

and causes this to be written to the TACL OUT file:

```
1 THE QUICK BROWN
3 THE LAZY DOG
```

VINSERT Command

Use the VINSERT command to insert lines from the current TACL IN file into a given line position in a variable.

```
VINSERT variable-level line-num
```

variable-level

is an existing variable level into which lines are to be inserted. It must not be in a shared segment, and must not be a DIRECTORY, a STRUCT, or a STRUCT item.

line-num

specifies the line number at which lines are to be inserted. It can be any of these:

F
L
number

F

specifies the first line in the variable level.

L

specifies a new line past the last line in the variable level.

number

is an integer identifying a specific line. It must be greater than zero and not greater than the number of lines in the variable level, plus one (it can specify a new line past the last existing line).

Considerations

- Inserted lines are placed just before the line specified by line-num.
- If the input text contains TACL metacharacters, the setting of the #INFORMAT built-in variable (for input to the IN file, including input to an interactive TACL process) or the ?FORMAT Directive on page 5-6 (for text in TACL programs) can affect how TACL interprets the string. For more information, see the #INFORMAT Built-In Variable on page 9-196 or the ?FORMAT directive.

Example

If the variable VAR has the contents shown:

```
THE QUICK BROWN
FOX JUMPED OVER
THE LAZY DOG
```

```
TWICE A DAY
EXCEPT TUESDAYS.
```

the command:

```
VINSERT var 3
```

causes TACL to prompt with line numbers that the inserted lines will have in the variable; TACL continues to accept lines until a line consisting solely of two slashes is entered. The insertion can also be ended by CTRL-y.

In this dialog at the current IN file, TACL prompts with line numbers and the user responds with text.

```
3 and over and
4 over and over
5 //
```

The variable then contains:

```
THE QUICK BROWN
FOX JUMPED OVER
and over and
over and over
THE LAZY DOG
TWICE A DAY
EXCEPT TUESDAYS.
```

VLIST Command

Use the VLIST command to list a range of lines in a variable.

```
VLIST [ / option [ , option ] / ... ] variable-level
  [ range ]
```

option

is any of these:

```
OUT file-name
QUIET
TO variable-level
```

`OUT` *file-name*

specifies a device, or a sequential file accessible to the sequential I/O (SIO) facility, that is to receive the listing. The listing includes line numbers. If you omit this option, TACL writes the listing to its current OUT file. If the QUIET option is present, this option is ignored.

If you specify an OUT file that does not exist, TACL creates an EDIT file named list-file. If you specify an OUT file that already exists, TACL appends the information to the end of the file.

`QUIET`

suppresses the listing, even if the OUT option is supplied.

`TO` *variable-level*

is the name of an existing variable level to which the listing is to be appended. Line numbers are not included. The QUIET option has no affect on this option.

variable-level

is an existing variable level containing the lines to be listed. It must not be in a shared segment, and must not be a DIRECTORY, a STRUCT, or a STRUCT item.

range

specifies the line or lines to be listed. If you omit it, all lines are listed. A range specification can be any of these:

```
A
line-num
line-num / line-num
```

A

specifies an individual line number. It can be any of these:

F
L
number

F

specifies the first line in the variable level.

L

specifies the last line in the variable level.

number

is an integer identifying a specific line.

Example

If these variables have the contents shown:

Var	Listvar
THE QUICK BROWN	12345678910
FOX JUMPED OVER	11121314151617
THE LAZY DOG	18192021
TWICE A DAY	
EXCEPT TUESDAYS	

the command:

```
VLIST /TO listvar/ var 2/4
```

causes them to contain:

Var	Listvar
THE QUICK BROWN	12345678910
FOX JUMPED OVER	11121314151617
THE LAZY DOG	18192021
TWICE A DAY	FOX JUMPED OVER
EXCEPT TUESDAYS	THE LAZY DOG
	TWICE A DAY

and causes this to be written to the TACL OUT file:

```
2 FOX JUMPED OVER
3 THE LAZY DOG
4 TWICE A DAY
```

VMOVE Command

Use the VMOVE command to delete a range of lines from one variable and insert them at a given line position in another variable.

```
VMOVE [ / option [ , option ] / ... ] source-var range
  dest-var dest-line
```

option

is any of these:

```
OUT file-name
QUIET
TO variable-level
```

OUT *file-name*

specifies a device, or a sequential file accessible to the sequential I/O (SIO) facility, that is to receive a listing of moved lines. The listing includes line numbers of the new lines in the destination variable. If you omit this option, TACL writes the listing to its current OUT file. If the QUIET option is present, this option is ignored.

If you specify an OUT file that does not exist, TACL creates an EDIT file named list-file. If you specify an OUT file that already exists, TACL appends the information to the end of the file.

QUIET

suppresses listing of moved lines even if the OUT option is supplied.

TO *variable-level*

is the name of an existing variable level to which a listing of all moved lines is to be appended. Line numbers are not included. The QUIET option has no affect on this option.

source-var

is an existing variable level from which lines are to be deleted. It must not be in a shared segment, and must not be a DIRECTORY, a STRUCT, or a STRUCT item.

range

specifies the line or lines to be moved. A range specification can be any of these:

```
A
line-num
line-num / line-num
```

A

specifies all lines in the variable level.

line-num

specifies an individual line number. It can be any of these:

```
F
L
number
```

F

specifies the first line in the variable level.

L

specifies the last line in the variable level.

number

is an integer identifying a specific line.

dest-var

is an existing variable level into which the moved lines are to be inserted. It must not be in a shared segment, and must not be a DIRECTORY, a STRUCT, or a STRUCT item.

dest-line

specifies the line number in the destination variable level at which the moved lines are to be inserted. This entry can be any of these:

```
F
L
number
```

F

specifies the first line in the variable level.

L

specifies a new line past the last line in the variable level.

number

is an integer identifying a specific line. It must be greater than zero and not greater than the number of lines in the variable level, plus one (it can specify a new line past the last existing line).

Considerations

- *source-var* and *dest-var* must not be the same variable level.
- Moved lines are inserted immediately preceding *dest-line*.

Example

If these variables have the contents shown:

Srcvar	Dstvar	Listvar
THE QUICK BROWN	ABCDEFG	12345678910
FOX JUMPED OVER	HIJKLMNOPQRST	1121314151617
THE LAZY DOG	UVWXYZ	18192021
TWICE A DAY		
EXCEPT		
TUESDAYS		

the command:

```
VMOVE /TO listvar/ srcvar 2/4 dstvar 3
```

causes them to contain:

Srcvar	Dstvar	Listvar
THE QUICK BROWN	ABCDEFG	12345678910
EXCEPT	HIJKLMNOPQRST	1121314151617
TUESDAYS	FOX JUMPED OVER	18192021
	THE LAZY DOG	FOX JUMPED OVER
	TWICE A DAY	THE LAZY DOG
	UVWXYZ	TWICE A DAY

and causes this to be written to the TACL OUT file:

```
3 FOX JUMPED OVER
4 THE LAZY DOG
5 TWICE A DAY
```

VOLUME Command

Use the VOLUME command to change temporarily your current settings for default volume and subvolume or to return to your saved defaults.

```
VOLUME [ [\node-name.] volume ] [ , "security" ]
```

\node-name

specifies your default system. If you omit \ node-name, your default system does not change.

volume

specifies your default volume and subvolume names. volume is one of these:

$*volume-name*
subvolume-name
$*volume-name*.*subvolume-name*

If you omit *volume-name* or *subvolume-name*, your logon default volume or subvolume becomes the current default volume or subvolume.

"security"

sets your current default file security. The system assigns this file security to newly created files unless you explicitly assign a different security when you create the file. Specify “security” as a four-character string “rwep” to specify the security for each type of file access: read, write, execute, and purge. For “rwep” you can specify A, G, O, N, C, or U. For more information about file security, see the *File Utility Program (FUP) Reference Manual*.

Considerations

- Settings made with the VOLUME command are in effect only temporarily-until you enter a LOGON command or a SYSTEM or VOLUME command with no parameters. For example, if you log on again, all your current defaults are reset to the original logon default system, volume, subvolume, and security (or as you specified in your last DEFAULT command).
- If the current default system is a remote system, you cannot specify a new current default volume name that has more than six characters after the dollar sign ($).
- Likewise, you cannot specify a new default system with the SYSTEM command if the current default volume name contains more than six characters after the dollar sign ($).
- Entering VOLUME with no parameters resets your current default system, volume, subvolume, and security to the values in effect at logon, or as they were set with your last DEFAULT command.

- To display your logon default volume and subvolume names, and your logon default security, enter the USERS program or use the WHO command.
- The security designations are:

O (Owner)	Only the owner can access the file; the owner must be logged onto the local system.
G (Group)	Anyone in the owner's group can access the file; the user must be logged onto the local system.
A (Anyone)	Any user can access the file; the user must be logged onto the local system.
U (User)	Only the owner can access the file; the owner may be logged onto the local system or a remote system.
C (Community)	Anyone in the owner's group can access the file; the user may be logged onto the local system or a remote system.
N (Network)	Any user can access the file; the user may be logged onto the local system or a remote system.
-	Only the local super ID can access the file.

- The security designation "-" (allow access to the local super ID only) is not permitted in a VOLUME command. Use #SET #PROCESSFILESECURITY instead.

Examples

1. To change your current default subvolume to $MANUF.BILLS, enter:

```
14> VOLUME $MANUF.BILLS
15>
```

2. You can reestablish the default system, volume, and subvolume that were in effect when you logged on (unless you subsequently changed your saved default setting with the DEFAULT program) by entering:

```
14> VOLUME
15>
```

VTREE Command

Use the VTREE command to list the names of all the variables in a directory, and in directories below it, and so on.

```
VTREE [ directory-name ]
```

Considerations

- If you omit directory-name, the home directory is assumed.
- One name is listed on each line.
- If a the name of a directory appears in the list, the names of the variables in it are indented on subsequent lines.

Example

This example is a sample VTREE listing:

```
40> VTREE

Directory :

MYSEG
  MYVAR
  ANOTHER^VAR
PROMPT^NEWPTIME
PROMPT^OLDPTIME
PROMPT^PROCESSID
UTILS
  TACL
    ACTIVATE
    ...
    YBUSDOWN
    ^UTILS
      A^UTIL
      ...
PROMPTER
_TACLBASE_FILE
41>
```

WAKEUP Command

You can use the WAKEUP command to set the TACL wakeup mode.

```
WAKEUP { ON | OFF }
```

ON

means that TACL is awakened (returned from the paused state) when any process started by TACL is deleted.

OFF

means that TACL is awakened only when the latest process started by TACL is deleted or when a process you designate in a PAUSE command is deleted. This is the logon default setting.

Consideration

A process is deleted when it abends, terminates normally, or a CPU fails. You can also delete a process by entering the STOP command. You can delete a TACL process with the EXIT command.

Example

To wake up TACL and return control of the terminal to TACL whenever a process that you started is deleted, enter:

```
26> WAKEUP ON
```

WHO Command

Use the WHO command to display information about your current TACL process.

```
WHO
```

Considerations

- The WHO command is useful for verifying your current defaults. For example, if you are on a network and use the SYSTEM command to access another system, you can issue a WHO command to check your defaults before you run programs or purge files. Checking defaults helps you avoid errors such as running programs on the wrong system or purging remote files instead of local files.
- The WHO command displays the name of the current default system if it is different from your local system.
- If the user logs on with a user name or user ID, Logon name shows *group.user* information. If the user logs on with an alias, Logon name shows *alias* information. For pre-D30 software RVUs, Logon name shows nothing.

Example

User MANUF.FRED, at \ROME, uses the SYSTEM command to make \NAPLES his current default system. When he enters WHO, this information is displayed:

```
12> WHO
Home terminal: $FRED
TACL process: \ROME.$C127
Primary CPU: 4 (TXP) Backup CPU: 5 (TXP)
Default Segment File: $DISK.#6726
  Pages allocated: 8 Pages Maximum: 1024
  Bytes Used: 10644 (0%) Bytes Maximum: 2097152
Current volume: $MORE.CHECK Current system: \NAPLES
Saved volume: $FRED.DATA
Userid: 8,44 Username: MANUF.FRED Security: "NUNU"
Default process: 4,167
```

or

```
12> WHO

Home terminal: $FRED
TACL process: \ROME.$C127
Primary CPU: 4 (TXP) Backup CPU: 5 (TXP)
Default Segment File: $DISK.#6726
Pages allocated: 8 Pages Maximum: 1024
Bytes Used: 10644 (0%) Bytes Maximum: 2097152
Current volume: $MORE.CHECK Current system: \NAPLES
Saved volume: $FRED.DATA
Userid: 8,44 Username: MANUF.FRED Security: "NUNU"
Default process: 4,167
Logon name: MYALIAS
```

This information is shown in this example:

- The name of the TACL home terminal.
- The process name of the current TACL. If the process is not named, its process ID (cpu,pin) is displayed.
- The CPU numbers and processor types of the current TACL primary and backup processes. If the primary TACL process is named but has no backup, the CPU number for the backup does not appear.
- Information about the TACL default segment file.
- The current default volume, subvolume, and system if remote.
- The saved default volume and subvolume as set by the last DEFAULTS command.
- The user ID including the user's group number and individual number.
- The user's name, including group name and the name by which the user is known.
- The current security setting to be assigned to files you create.
- The default process if there is one.

XBUSDOWN/YBUSDOWN Command (Super-Group Only)

Use the XBUSDOWN or YBUSDOWN command to inform the operating system that an interprocessor bus is being brought down and should not be used. This command is equivalent to the BUSCMD process. The system reports the results of an XBUSDOWN or YBUSDOWN command at the operator console. To use the XBUSDOWN or YBUSDOWN command, you must have a group ID of 255.

```
{ X | Y }BUSDOWN from-cpu , to-cpu
```

`X | Y`

specifies the bus (X or Y) to be brought down.

`from-cpu` and *`to-cpu`*

are CPU numbers in the range from 0 through 15. Each CPU number specifies one endpoint of a segment of the designated bus. On this segment, transfers are not permitted. Specify -1 to indicate all processors.

Example

In a four-processor system, a super-group user can bring down the X bus from processor 1 to all other processors by entering:

```
13> XBUSDOWN 1, -1
THE X BUS FROM CPU 01 TO 00 HAS BEEN DOWNED.
THE X BUS FROM CPU 01 TO 01 HAS BEEN DOWNED.
THE X BUS FROM CPU 01 TO 02 HAS BEEN DOWNED.
THE X BUS FROM CPU 01 TO 03 HAS BEEN DOWNED.
```

XBUSUP/YBUSUP Command (Super-Group Only)

Use the XBUSUP or YBUSUP command to inform the operating system that an interprocessor bus is available for use. This command is equivalent to the BUSCMD process. The system reports the results of an XBUSUP or YBUSUP command at the operator console. To use the XBUSUP or YBUSUP command, you must have a group ID of 255.

```
{ X | Y }BUSUP from-cpu , to-cpu
```

X | Y

specifies the bus (X or Y) to be brought up.

from-cpu and *to-cpu*

are CPU numbers in the range from 0 through 15. Each CPU number specifies the endpoint of a segment of the designated bus. On this segment, transfers are now permitted. Specify -1 to indicate all processors.

Example

A super-group user can bring the Y bus back up from processor 1 to processor 2 by entering:

```
15> YBUSUP 1,2
THE Y BUS FROM CPU 01 TO 02 HAS BEEN UPPED.
```

Exclamation Point (!) Command

Use the exclamation point (!) command to reexecute immediately a previous command line, without modifications.

```
!
 [num]
 [- num]
 [text]
```

num

is an absolute history number.

- num

is a number relative to the current history number.

text

is a text string.

Considerations

- If you enter nothing but !, TACL reexecutes the previous command (! is the same as ! - 1).
- If TACL cannot find a command that matches your specification, whether by absolute history number, relative history number, or command text, it issues an error message.
- You must enter the ! command from the IN file (normally your home terminal); you cannot include it in a macro, for example. Similarly, you cannot change ! to another name with an ALIAS, nor can you program a function key to execute the ! command.

Examples

1. This example illustrates the use of the ! command to recall and reexecute the coding at history number 5:

```
10> !5
10> OUTVAR edstat
STATUS *,PROG $SYSTEM.SYSTEM.TEDIT
```

2. This example shows the use of the ! command to reexecute the most recent occurrence of a SET command:

```
11> !SET
11> SET VARIABLE edstat STATUS *,PROG $SYSTEM.SYSTEM.TEDIT
```

Question Mark (?) Command

Use the question mark (?) command to display a previous command line.

```
?
 [num]
 [- num]
 [text]
```

num

is an absolute history number.

- num

is a number relative to the current history number.

text

is a text string.

Considerations

- If you enter a question mark (?) without an argument, TACL displays the previous command (? is the same as ? - 1).
- If TACL cannot find a command that matches your specification-whether by absolute history number, relative history number, or command text-it issues an error message.
- You must enter the ? command from the IN file (normally your home terminal); you cannot include it in a macro, for example. Similarly, you cannot change ? to another name with an ALIAS, nor can you program a function key to execute the ? command.
- The ? command does not increment the history number in the TACL prompt.

Examples

1. In this example, the user first uses the ? command to determine what command was issued at history number 5. The user then enters the ! command to reexecute that command (OUTVAR edstat).

```
10> ?5
10> OUTVAR edstat
10> !5
STATUS *,PROG $SYSTEM.SYSTEM.TEDIT
```

2. This example shows the use of the ? command to see the most recent command that started with the text string “SET.” The user then enters the ! command to reexecute the last SET command.

```
11> ?SET
11> SET VARIABLE edstat STATUS *,PROG $SYSTEM.SYSTEM.TEDIT
11> !SET
11> SET VARIABLE edstat STATUS *,PROG $SYSTEM.SYSTEM.TEDIT
```

9

Built-In Functions and Variables

This section contains an alphabetic summary of all built-in functions and variables, followed by descriptions, in alphabetic order, of the syntax for each built-in function and built-in variable. Each description (where appropriate) contains:

- A summary of the purpose of the function or variable
- The syntax of the function or variable, including a description of the syntax of parameters (in the case of a built-in variable, the syntax of the #SET function used to assign values to the variable is also described)
- The result returned from a function or variable invocation
- The listing format used for output (if a function produces a display or listing output)
- Considerations for the use of the function or variable
- Examples of usage of the function or variable

Note. All examples in this section are based on the assumptions that the built-in variable #INFORMAT has been set to TACL, which enables recognition and processing of the TACL special characters ([and], for example); that the built-in variable #PMSEARCHLIST has been set to include (at least) $SYSTEM.SYSTEM and the keyword #DEFAULTS, which enables the use of implied RUN commands; and that the required TACL library files have been loaded into memory.

Many TACL built-in functions access Guardian procedures to provide functionality. Appendix C, Mapping TACL Built-In Functions to Guardian Procedures, shows whether a TACL built-in function accesses a Guardian procedure, and if so, which procedure or procedures. To enhance your understanding of a built-in function, refer to the description for the corresponding Guardian procedure in the *Guardian Procedure Calls Reference Manual*.

Summary of Built-In Functions

Table 9-1 on page 9-2 summarizes the built-in functions and their uses.

Note. All TACL built-ins are executed by the TACL process, which runs only on the node where it was started, regardless of any SYSTEM commands that are issued. To execute a built-in command on another system, you must start a new TACL process on that system.

Table 9-1. Built-In Functions (page 1 of 8)

Function	Description
#ABEND Built-In Function	Immediately terminates a process
#ABORTTRANSACTION Built-In Function	Aborts and backs out a transaction
#ACTIVATEPROCESS Built-In Function	Returns process or process pair from suspended state to ready state
#ADDDSTTRANSITION Built-In Function (Super-Group Only)	Adds entry to daylight savings time transition table
#ALTERPRIORITY Built-In Function	Changes execution priority of a process or process pair
#APPEND Built-In Function	Appends additional lines to a variable level
#APPENDV Built-In Function	Appends a line from one variable level to another
#ARGUMENT Built-In Function	Parses arguments to routines
#BACKUPCPU Built-In Function	Sets or changes the TACL backup CPU
#BEGINTRANSACTION Built-In Function	Starts a new transaction
#BREAKPOINT Built-In Function	Sets or deletes _DEBUGGER breakpoint for a specific variable level
#BUILTINS Built-In Function	Examines names of TACL built-in functions and variables
#CASE Built-In Function	Chooses one out of a set of options
#CHANGEUSER Built-In Function	Logs user on under different user ID
#CHARADDR Built-In Function	Converts line address to character address
#CHARBREAK Built-In Function	Inserts line break in variable at character address
#CHARCOUNT Built-In Function	Obtains number of characters in variable
#CHARDEL Built-In Function	Deletes characters from variable at character address
#CHARFIND Built-In Function	Locates text in variable, searching forward from character address
#CHARFINDR Built-In Function	Locates text in variable, searching backward from character address
#CHARFINDRV Built-In Function	Locates string in variable, searching backward from character address
#CHARFINDV Built-In Function	Locates string in variable, searching forward from character address
#CHARGET Built-In Function	Obtains copy of specified number of characters from a variable
#CHARGETV Built-In Function	Copies specified number of characters from one variable to another

Table 9-1. Built-In Functions (page 2 of 8)

Function	Description
#CHARINS Built-In Function	Inserts text into a variable at character address
#CHARINSV Built-In Function	Inserts string into a variable at character address
#COLDLOADTACL Built-In Function	Determines if TACL process is the “cold-load TACL”
#COMPAREV Built-In Function	Compares one variable with another
#COMPUTE Built-In Function	Returns value of expression
#COMPUTEJULIANDAYNO Built-In Function	Converts Gregorian calendar date to a Julian day number
#COMPUTETIMESTAMP Built-In Function	Converts calendar date to a four-word timestamp
#COMPUTETRANSID Built-In Function	Converts separate components of a transaction ID to one numeric transaction ID
#CONTIME Built-In Function	Converts timestamp to seven-digit date and time
#CONVERTPHANDLE Built-In Function	Converts a process file identifier to a process handle, or vice versa
#CONVERTPROCESSTIME Built-In Function	Converts time value obtained by PROCESSTIME option of #PROCESSINFO
#CONVERTTIMESTAMP Built-In Function	Converts GMT timestamp to a local-time-based timestamp, or a local-time-based timestamp to a GMT timestamp
#CREATEFILE Built-In Function	Creates a file
#CREATEPROCESSNAME Built-In Function	Creates unique process name
#CREATEREMOTENAME Built-In Function	Returns process name unique to specified system
#DEBUGPROCESS Built-In Function	Calls debugger for specified process
#DEF Built-In Function	Defines a variable
#DEFINEADD Built-In Function	Adds a DEFINE to TACL context, using attributes in the working set
#DEFINEDELETE Built-In Function	Deletes a DEFINE from TACL context
#DEFINEDELETEALL Built-In Function	Deletes all DEFINEs from TACL context
#DEFINEINFO Built-In Function	Gets information about a DEFINE
#DEFINENAMES Built-In Function	Gets names of all DEFINEs that match specified template
#DEFINENEXTNAME Built-In Function	Gets name of next DEFINE following specified DEFINE in sequence established by the operating system

Table 9-1. Built-In Functions (page 3 of 8)

Function	Description
#DEFINEREADATTR Built-In Function	Gets value of specified attribute
#DEFINERESTORE Built-In Function	Creates or replaces active DEFINE, or replaces working set with contents of DEFINE previously saved with #DEFINESAVE
#DEFINERESTOREWORK Built-In Function	Restores DEFINE working set from background set
#DEFINESAVE Built-In Function	Saves copy of active DEFINE or working set for later restoration with #DEFINERESTORE
#DEFINESAVEWORK Built-In Function	Saves DEFINE current working set to background set
#DEFINESETATTR Built-In Function	Modifies value of specified DEFINE attribute in current working set
#DEFINESETLIKE Built-In Function	Initializes current working set with attributes of an existing DEFINE
#DEFINEVALIDATEWORK Built-In Function	Checks DEFINE current working set for consistency
#DELAY Built-In Function	Causes TACL to wait for specified time
#DELTA Built-In Function	Acts as a complex character processor
#DEVICEINFO Built-In Function	Gets detailed information about a device
#EMPTY Built-In Function	Determines whether specified string contains text
#EMPTYV Built-In Function	Determines whether a variable level contains any lines
#EMSADDSUBJECT Built-In Function	Adds subject token to event message buffer
#EMSADDSUBJECTV Built-In Function	Adds subject token to event message buffer, obtaining token values from a STRUCT
#EMSGET Built-In Function	Retrieves token values from SPI buffer
#EMSGETV Built-In Function	Copies token values from SPI buffer to a STRUCT
#EMSINIT Built-In Function	Initializes a STRUCT as event message buffer
#EMSINITV Built-In Function	Initializes STRUCT as event message buffer, obtaining initial values from another STRUCT
#EMSTEXT Built-In Function	Converts information from event buffer to printable text
#EMSTEXTV Built-In Function	Converts information from event buffer to printable text, copies text to a STRUCT
#ENDTRANSACTION Built-In Function	Commits data base changes associated with a transaction
#EOF Built-In Function	Sets flag so that a process receives an end-of-file after reading all data in a variable

Table 9-1. Built-In Functions (page 4 of 8)

Function	Description
#ERRORTEXT Built-In Function	Used with exception handlers to catch error text
#EXCEPTION Built-In Function	Determines why a routine was invoked during exception handling
#EXTRACT Built-In Function	Deletes first line of a variable level
#EXTRACTV Built-In Function	Moves first line of a variable level to another variable
#FILEGETLOCKINFO Built-In FunctionO	Obtains information about record locks and file locks
#FILEINFO Built-In Function	Gets information about a file
#FILENAMES Built-In Function	Lists file names
#FILTER Built-In Function	Indicates which exceptions a routine can handle
#FRAME Built-In Function	Tracks pushed variables
#GETCONFIGURATION Built-In Function	Obtains settings of flags that affect TACL behavior
#GETPROCESSSTATE Built-In Function	Obtains process state information about the current TACL process
#GETSCAN Built-In Function	Obtains number of characters passed over by #ARGUMENT
#HISTORY Built-In Function	Operates on commands in history buffer
#IF Built-In Function	Executes one of two options
#INITTERM Built-In Function	Resets your home terminal to its configured settings
#INLINEEOF Built-In Function	Sends end-of-file to process running under control of INLINE facility
#INPUT Built-In Function	Reads information from TACL primary input file
#INPUTV Built-In Function	Reads information from TACL primary input file into a variable level
#INTERACTIVE Built-In Function	Determines whether your TACL is interactive
#INTERPRETJULIANDAYNO Built-In Function	Converts Julian day number to year, month, and day
#INTERPRETTIMESTAMP Built-In Function	Breaks down four-word timestamp to its component parts
#INTERPRETTRANSID Built-In Function	Converts numeric transaction ID to its separate component values
#JULIANTIMESTAMP Built-In Function	Obtains four-word timestamp
#KEEP Built-In Function	Removes all but specified level of a variable
#KEYS Built-In Function	Displays defined function keys
#LINEADDR Built-In Function	Converts character address to line address

Table 9-1. Built-In Functions (page 5 of 8)

Function	Description
#LINEBREAK Built-In Function	Inserts line break in variable at line address
#LINECOUNT Built-In unction	Obtains number of lines in a variable
#LINEDEL Built-In Function	Deletes lines from variable at line address
#LINEFIND Built-In Function	Locates text in variable, searching forward from line address
#LINEFINDR Built-In Function	Locates text in variable, searching backward from line address
#LINEFINDRV Built-In Function	Locates string in variable, searching backward from line address
#LINEFINDV Built-In Function	Locates string in variable, searching forward from line address
#LINEGET Built-In Function	Gets copy of specified number of lines from a variable
#LINEGETV Built-In Function	Copies specified number of lines from one variable to another
#LINEINS Built-In Function	Inserts text into a variable at line address
#LINEINSV Built-In Function	Inserts string into a variable at line address
#LINEJOIN Built-In Function	Deletes line break at end of a line, joining following line to it
#LOAD Built-In Function	Processes a TACL library file
#XLOADEDFILES Built-In Function	Gets information about all LOADFILES associated with a given process
#LOCKINFO Built-In Function	Gets information about record locks
#LOGOFF Built-In Function	Logs off current TACL
#LOOKUPPROCESS Built-In Function	Gets information about a PPD entry
#LOOP Built-In Function	Repeatedly executes one or more statements in a function
#MATCH Built-In Function	Determines whether given string satisfies a template
#MOM Built-In Function	Obtains identity of creator process
#MORE Built-In Function	Determines whether an entire argument has been consumed
#MYGMOM Built-In Function	Obtains identity of TACL job ancestor process
#MYPID Built-In Function	Obtains your CPU,PIN number
#MYSYSTEM Built-In Function	Determines name of system executing current TACL
#NEWPROCESS Built-In Function	Starts a process
#NEXTFILENAME Built-In Function	Determines file following specified file

Table 9-1. Built-In Functions (page 6 of 8)

Function	Description
#OPENINFO Built-In Function	Gets information about file openers
#OUTPUT Built-In Function	Writes data to an output file
#OUTPUTV Built-In Function	Writes contents of a variable level to an output file
#PAUSE Built-In Function	Gives control of your terminal to another process
#POP Built-In Function	Deletes top level of variables
#PROCESS Built-In Function	Obtains identity of last process created or paused for by TACL
#PROCESSEXISTS Built-In Function	Determines whether a process exists
#PROCESSINFO Built-In Function	Requests information about a process
#PROCESSLAUNCH Built-In Function	Starts a process
#PROCESSORSTATUS Built-In Function	Determines status of 16 possible CPUs on a given system
#PROCESSORTYPE Built-In Function	Determines processor type of given system or process
#PURGE Built-In Function	Deletes a file
#PUSH Built-In Function	Creates new top level for variables
#RAISE Built-In Function	Defines exception to be filtered by routine
#RENAME Built-In Function	Changes name of existing disk file
#REPLY Built-In Function	Adds text to reply if TACL IN file is $RECEIVE
#REPLYV Built-In Function	Adds copy of text from variable to reply if TACL IN file is $RECEIVE
#REQUESTER Built-In Function	Reads from and writes to files
#RESET Built-In Function	Sets argument pointer, frame pointer, reply value, and result text
#REST Built-In Function	Obtains remaining argument string for current routine
#RESULT Built-In Function	Supplies text that replaces original invocation of routine
#RETURN Built-In Function	Exits from a routine immediately
#ROUTINENAME Built-In Function	Obtains name of variable in which containing routine resides
#SEGMENT Built-In Function	Obtains name of segment file that TACL is using for its variables
#SEGMENTCONVERT Built-In Function	Converts segment file between C00/C10 format and newer format
#SEGMENTINFO Built-In Function	Gets information about segments being used by TACL

Table 9-1. Built-In Functions (page 7 of 8)

Function	Description
#SEGMENTVERSION Built-In Function	Determines whether segment file is C00/C10 format or newer format
#SERVER Built-In Function	Creates and deletes servers
#SET Built-In Function	Puts data in a variable level
#SETBYTES Built-In Function	Copies as many bytes as can fit from one STRUCT or STRUCT item to another
#SETCONFIGURATION Built-In Function	Sets the configuration of the running TACL process
#SETMANY Built-In Function	Sets more than one variable level
#SETPROCESSSTATE Built-In Function	Sets a process state flag for the current TACL process
#SETSCAN Built-In Function	Indicates position at which next #ARGUMENT is to resume processing arguments
#SETSYSTEMCLOCK Built-In Function (Super-Group Only)	Changes setting of system clock
#SETV Built-In Function Use	Copies one variable level to another
#SHIFTSTRING Built-In Function	Changes text from uppercase to lowercase or from lowercase to uppercase
#SORT Built-In Function	Sorts a list of text
#SPIFORMATCLOSE Built-In Function	Closes an open EMS formatter template file
#SSGET Built-In Function	Retrieves binary token values from an SPI buffer and returns an external representation of those values
#SSGETV Built-In Function	Copies binary token values from an SPI buffer to a STRUCT
#SSINIT Built-In Function	Prepares a STRUCT for use with other #SSxxx built-in functions
#SSMOVE Built-In Function	Copies tokens from one SPI buffer to another
#SSNULL Built-In Function	Initializes extensible structured tokens
#SSPUT Built-In Function	Converts external representation of token values to binary form, puts them in SPI buffer
#SSPUTV Built-In Function	Copies binary token values from a variable level into an SPI buffer
#STOP Built-In Function	Terminates a process
#SUSPENDPROCESS Built-In Function	Suspends a process
#SWITCH Built-In Function	Switches TACL to its backup process
#SYSTEM Built-In Function	Temporarily changes your default system

Table 9-1. Built-In Functions (page 8 of 8)

Function	Description
#SYSTEMNAME Built-In Function	Requests a system by name
#SYSTEMNUMBER Built-In Function	Requests a system by number
#TACLOPERATION Built-In Function	Determines whether TACL is reading commands from IN or $RECEIVE
#TACLVERSION Built-In Function	Obtains TACL product number
#TIMESTAMP Built-In Function	Obtains internal form of CPU interval clock
#TOSVERSION Built-In Function	Obtains current RVU of the operating system
#UNFRAME Built-In Function	Pops all variables pushed since #FRAME
#USERID Built-In Function	Specifies a user by user ID
#USERNAME Built-In Function	Specifies a user by user name
#VARIABLEINFO Built-In Function	Gets information about a variable
#VARIABLES Built-In Function	Obtains names of all variables in your home directory
#VARIABLESV Built-In Function	Obtains names of all your variables, puts them on separate lines in an existing variable level
#WAIT Built-In Function	Specifies variables for which a routine must wait
#XFILEINFO Built-In Function	Implements FILEINFO command
#XFILENAMES Built-In Function	Implements FILENAMES command
#XFILES Built-In Function	Implements FILES command
#XLOGON Built-In Function	Implements LOGON command
#XPPD Built-In Function	Implements PPD command
#XSTATUS Built-In Function	Implements STATUS command

Summary of Built-In Variables

You can use built-in variables anywhere a built-in function can be used and as the object of #PUSH, #POP, and #SET. In all cases, #PUSH saves a copy of the variable, #SET assigns a new value to it, and #POP removes that value, restoring the variable to the value last saved by #PUSH.

Table 9-2 on page 9-10 summarizes the built-in variables and the data they represent.

Note. All TACL built-ins are executed by the TACL process, which runs only on the node where it was started, regardless of any SYSTEM commands that are issued. To execute a built-in command on another system, you must start a new TACL process on that system.

Table 9-2. Built-In Variables (page 1 of 2)

Variable	Description
#ASSIGN Built-In Variable	Holds information about all currently defined unit names
#BREAKMODE Built-In Variable	Affects BREAK key operation
#CHARACTERRULES Built-In Variable	Holds name of current character-processing rules file
#DEFAULTS Built-In Variable	Holds volume defaults you set
#DEFINEMODE Built-In Variable	Holds flag indicating whether DEFINEs can be used
#ERRORNUMBERS Built-In Variable	Holds information about the latest error detected by the current TACL process
#EXIT Built-In Variable	Holds state of exit flag
#HELPKEY Built-In Variable	Holds name of current help key
#HIGHPIN Built-In Variable	Specifies the default PIN range for new processes
#HOME Built-In Variable	Represents your home directory
#IN Built-In Variable	Holds name of IN file used by TACL
#INFORMAT Built-In Variable	Represents formatting mode for #INPUT
#INPUTEOF Built-In Variable	Holds state of INPUTEOF flag
#INLINEECHO Built-In Variable	Controls whether TACL echoes to its OUT file lines passed as input to inline processes
#INLINEOUT Built-In Variable	Controls whether TACL copies to its own OUT file lines written by inline processes to their OUT files
#INLINEPREFIX Built-In Variable	Holds prefix used to identify lines to be passed to inline processes instead of being acted upon by TACL
#INLINEPROCESS Built-In Variable	Holds process ID of current inline process, if such exist
#INLINETO Built-In Variable	Holds name of variable, if any, to which TACL appends lines written by inline processes to their OUT files
#INSPECT Built-In Variable	Holds state of INSPECT flag
#MYTERM Built-In Variable	Holds name of your home terminal
#OUT Built-In Variable	Holds name of OUT file used by TACL

Table 9-2. Built-In Variables (page 2 of 2)

Variable	Description
#OUTFORMAT Built-In Variable	Represents formatting mode for #OUTPUT
#PARAM Built-In Variable	Holds list of all your parameters, or a specified parameter
#PMSEARCHLIST Built-In Variable	Holds list of subvolumes to be searched for program and macro files
#PMSG Built-In Variable	Holds state of PMSG flag
#PREFIX Built-In Variable	Holds contents of prefix string
#PROMPT Built-In Variable	Represents state of prompt flag
#REPLYPREFIX Built-In Variable	Holds value of your reply prefix
#ROUTEPMSG Built-In Variable	Suppresses the outputting of system and process messages
#SHIFTDEFAULT Built-In Variable	Holds #SHIFTSTRING default
#TACLSECURITY Built-In Variable	Represents TACL security
#TRACE Built-In Variable	Represents state of TRACE flag
#USELIST Built-In Variable	Holds your use list
#WAKEUP Built-In Variable	Represents setting of WAKEUP flag
#WIDTH Built-In Variable	Holds value of width register

Built-In Function and Variable Descriptions

The remainder of this section lists built-in functions and variables, in alphabetic order.

#ABEND Built-In Function

Use #ABEND to request that a process terminate immediately; you can also specify values for fields in the resulting process deletion message.

```
#ABEND [ / option [ , option ] ... / ]
[ [\node-name.]{$process-name | cpu,pin } [ text ] ]
```

option

is any of these:

COMPLETIONCODE *num*

specifies the completion code to be returned in the process deletion message; num is a signed integer from -32768 to +32767. Numbers from -32768 to -1 are reserved for use in the TACL software product. Numbers from 0 to 999 are reserved for shared use by the TACL software product and the user. Numbers from 1000 to 32767 are reserved for use by customers. See the *Guardian Procedure Calls Reference Manual* for a list of defined completion codes.

ERROR

changes the behavior of #ABEND as described under Result on page 9-13.

SUBSYS *ssid*

specifies the subsystem for distributed systems management applications; see the *TACL Programming Guide* for details on subsystem IDs.

TERMINATIONINFO *num*

specifies information about the termination; num is a signed integer in the range -32768 to +32767.

node-name

is the system where the process to be deleted resides.

$*process-name*

is the name of the process or process pair.

cpu,pin

is the CPU number and process number for the process.

text

is text, 0 to 80 characters, to be included in the process deletion message. Leading and trailing spaces are ignored.

Result

- If you do not specify the ERROR option, #ABEND returns -1 if it is successful, 0 if it is not.
- If you include the ERROR option , #ABEND returns the file-system error code passed to it by the ABEND operating system procedure. In this case, zero indicates no error.
- If the #ABEND function terminates the TACL process from which it was issued, #ABEND returns nothing.

Considerations

- If the process cannot be terminated immediately, the ABEND operating system procedure queues the request.
- If you do not specify a process, #ABEND terminates the last process started by TACL or the process for which TACL most recently paused, if that process is still running. If there is no default process, you must include the process specification.
- You can include the text argument only when a process specification is present.
- COMPLETIONCODE, SUBSYS, TERMINATIONINFO, and text are ignored, except when terminating your current TACL process (or, if it is a process pair, either its primary or backup process). If you are abending your TACL, those items you specify are included in the “abend” system message.

Examples

1. In this macro, TACL terminates $TMP2 after starting it:

```
?SECTION job^mgr ROUTINE
#FRAME
#PUSH errpm
== Start $TMP2
FUP /NOWAIT, NAME $TMP2/
== Now stop $TMP2:
#OUTPUT Error occurred, abending $TMP2...
#SET errpm [#ABEND /COMPLETIONCODE 20, ERROR/ $TMP2]
#OUTPUT Diagnostics: errpm = [errpm]
#UNFRAME
```

2. This routine produces this output after loading and invoking job^mgr:

```
14> job^mgr
Error occurred, abending $TMP2...
Diagnostics: errpm = 0
```

```
ABENDED: $TMP2
CPU Time 0:00:00.018
Completion Code 20
Termination Info 2278
Subsystem
```

#ABORTTRANSACTION Built-In Function

Use #ABORTTRANSACTION to abort and back out a transaction. This function invokes the ABORTTRANSACTION operating system procedure. When the process that issued #BEGINTRANSACTION (or its backup) calls this procedure, the TMF subsystem backs out the database changes made for the current transaction identifier of the process.

```
#ABORTTRANSACTION
```

Result

#ABORTTRANSACTION returns zero if successful, or a file-system error indicating the reason the operation failed.

Consideration

To commit the database changes associated with a transaction identifier, use the #ENDTRANSACTION built-in function.

#ACTIVATEPROCESS Built-In Function

Use #ACTIVATEPROCESS to restart a suspended process or process pair. This function invokes the ACTIVATEPROCESS operating system procedure.

```
#ACTIVATEPROCESS [ [\node-name.]{$process-name | cpu,pin} ]
```

\node-name

is the system where the process resides.

$process-name

is the name of the process or process pair.

cpu,pin

is the CPU number and process number for the process.

Result

#ACTIVATEPROCESS returns a nonzero integer if it is successful; otherwise, it returns zero.

Considerations

- If you omit the process specification, #ACTIVATEPROCESS activates the default process (the last process started by the current TACL, or the one for which TACL most recently paused), if that process is still running.
- To find the identity of the default process, use the #PROCESS built-in function.

#ADDDSTTRANSITION Built-In Function (Super-Group Only)

Use #ADDDSTTRANSITION to add an entry to the daylight savings time transition table.

```
#ADDDSTTRANSITION low-gmt high-gmt offset
```

low-gmt

is an integer (see #COMPUTETIMESTAMP Built-In Function on page 9-73 for its format) that represents the Greenwich mean time when offset first applies.

high-gmt

is an integer that represents the GMT when offset no longer applies.

offset

is an integer that represents the number of seconds to be added to the local standard time to produce the local civil time.

Result

#ADDDSTTRANSITION returns a nonzero value if it is successful, zero otherwise.

Considerations

- To use the #ADDDSTTRANSITION function, you must have a group ID of 255.
- The DST transition table must be loaded in time sequence with no gaps: in each entry after the first, *low-gmt* must be the same as the *high-gmt* in the preceding #ADDDSTTRANSITION call.
- Daylight saving time information is represented as the variable DAYLIGHT_SAVINGS_TIME in the CONFTEXT file used as part of system generation. There are three possible values: NONE (do not adjust the time), USA66 (automatically adjust the time when appropriate), and TABLE. The default value is NONE. NONE and USA66 require no further user input. TABLE requires subsequent manipulation of the daylight savings time transition table (DST).
  - On D-series RVU systems, the value to be assigned to DAYLIGHT_SAVING_TIME is set by editing the CONFTEXT configuration file used during system generation.
  - On G-series RVU systems, the value to be assigned to DAYLIGHT_SAVING_TIME is set with the Subsystem Control Facility (SCF) for the Kernel Subsystem (rather than in the CONFTEXT file).

  If TABLE is specified you must be a super-group user (255, *n*) to manipulate entries in the DST. When TABLE is specified:

- For D-series RVUs, manipulate the DST either interactively using the ADDDSTTRANSITION TACL command or programmatically using the ADDDSTTRANSITION Guardian procedure.
- For G04.00 and earlier G-series RVUs, manipulate the DST either interactively using the ADDDSTTRANSITION TACL command or the SCF ALTER command for the Kernel Subsystem, or programmatically using the ADDDSTTRANSITION Guardian procedure.
- For G05.00 and later G-series RVUs, DST_TRANSITION_ADD_ supersedes the ADDDSTTRANSITION Guardian procedure. The TACL product, however, continues to access the ADDDSTTRANSITION Guardian procedure.

For more information about setting the input variable values when TABLE is specified, refer to the description of the COMPUTETIMESTAMP procedure in the *Guardian Procedure Calls Reference Manual*.

#ALTERPRIORITY Built-In Function

Use #ALTERPRIORITY to change the priority of a process or process pair. This function invokes the ALTERPRIORITY operating system procedure.

```
#ALTERPRIORITY [ [\node-name.]{$process-name | cpu,pin} ] pri
```

\node-name

is the system where the process resides.

$process-name

is the name of the process or process pair.

cpu,pin

is the CPU number and process number for the process.

pri

is the new execution priority of the process. Specify pri as an integer in the range 1 to 199. (Processes with higher numbers are executed first.)

Result

#ALTERPRIORITY returns a nonzero integer if it is successful; otherwise, it returns zero.

Considerations

- If you do not specify a process, #ALTERPRIORITY changes the priority of the process last started by the current TACL or for which TACL most recently paused, if that process still exists. That process is called the default process. To retrieve the default process, use the #PROCESS built-in function. If there is no default process, you must include a process specification or an error will result.
- The super ID can change the priority of any process in the system.
- A group manager can alter the priority of any process whose process accessor ID matches any user ID in the group.
- Standard users can change the priority of only those processes whose process accessor IDs match their user ID. (For a description of process accessor IDs and creator accessor IDs, see the *Guardian User's Guide*.)
- Before increasing the priority of a process, carefully consider the effect the change might have on system performance. For example, assigning a high priority to CPU-bound processes, such as those involving lengthy arithmetic computations, can significantly degrade system performance.
- If a CMON process is running, it can affect the range of priorities you can specify. For more information, see the RUN[D|V] Command on page 8-156.

#APPEND Built-In Function

Use #APPEND to add a line of text to the end of a variable level.

```
#APPEND to-variable-level [ text ]
```

to-variable-level

is the name of an existing variable level to which text is to be appended.

text

is the text to be added. If you omit text, a blank line is added to the end of the variable level.

Result

#APPEND returns nothing.

Considerations

- The appended text always starts a new line; #APPEND cannot append text to an existing line.
- If text is a variable name in brackets, and that variable contains multiple lines, enclose the entire function call in square brackets. Otherwise, TACL tries to execute the second line.

Example

This text defines a variable called kingandi, and then appends “Please read the attached message” to the initial text:

```
11> [#DEF kingandi TEXT |BODY|Attention managers]
12> #APPEND kingandi Please read the attached message
13> #OUTPUTV kingandi
Attention managers
Please read the attached message
14>
```

#APPENDV Built-In Function

Use #APPENDV to append a string to the contents of a variable level, to append a copy of the contents of one variable level to the end of another variable level, or to send data to a file or process opened by #REQUESTER.

```
#APPENDV to-variable-level { from-variable-level | string }
```

to-variable-level

is the name of an existing variable level whose definition is to be modified.

from-variable-level

is the name of an existing variable level whose definition is to be copied. This variable level is not modified.

string

is an argument of type STRING.

Result

#APPENDV returns nothing.

Considerations

- If *from-variable-level* is a variable level of type STRUCT, the #APPENDV function appends a line containing the binary data of the STRUCT to *to-variable-level*, which can be of another type. This function is primarily used to supply structured data to variables associated with servers and requesters.
- If *from-variable-level* is empty, #APPENDV does nothing.
- The form #APPENDV var struct is not equivalent to #APPEND var [struct], which appends the STRUCT external representation, rather than its binary data.
- If you are using #APPENDV to access data in a file, check the error variable (defined with #REQUESTER) to see if a file system error occurred.

Example

Assuming the home terminal name is $MINE, the following:

```
#PUSH sayterm termname
#SETV sayterm "My terminal is"
#SET termname [#MYTERM]
#APPENDV sayterm termname '+' " at this time."
```

causes SAYTERM to contain “My terminal is $MINE at this time.”

#ARGUMENT Built-In Function

Use #ARGUMENT to parse the arguments passed to a routine. #ARGUMENT returns the position number (in the list of alternatives to #ARGUMENT) of the alternative that matches the type of the routine argument being examined. You can specify up to eight alternatives to #ARGUMENT.

Each call to #ARGUMENT implicitly moves an internal pointer (called the current position) through the list of the routine arguments, unless the PEEK option is used.

The current position can be obtained by #GETSCAN and can be changed with #SETSCAN or with the START option of the CHARACTERS alternative.

Alternatives are tested, in the order supplied, against the routine argument at the current position. When an alternative is satisfied, no further alternatives are tried. If none of the supplied alternatives can be satisfied, the TACL error handler takes over.

```
#ARGUMENT [ / option [ , option ] ... / ]
  alternative [ alternative ] ...
```

option

is any of these:

PEEK

prevents #ARGUMENT from moving the current position. The operation of #ARGUMENT is normal in all other ways; in particular, if no matching type is supplied, an error is still generated.

TEXT *variable-level*

specifies the name of an existing variable level that is to receive an exact copy (except that leading and trailing spaces and blank lines are suppressed) of the portion of the argument sequence that was processed in the call to #ARGUMENT.

If the call to the routine is issued from an IN file or is loaded under control of a ?FORMAT directive in a library file, the setting of the #INFORMAT built-in variable affects how TACL copies the text. If, for example, TACL is using PLAIN format, TACL does not recognize square brackets as metacharacters.

VALUE *variable-level*

specifies the name of an existing variable level that is to receive the TACL interpretation of the portion of the argument sequence that was processed in the call to #ARGUMENT. This interpretation can be different from what is obtained by the TEXT option, depending on the argument type and its options.

If the call to the routine is issued from an IN file or is loaded under control of a ?FORMAT directive in a library file, the setting of the #INFORMAT built-in variable affects how TACL interprets the argument portion. If, for example,

TACL is using PLAIN format, TACL does not recognize square brackets as metacharacters.

The identity of the alternative being tested can have an effect on the result of the VALUE option. For example, if there is a variable FN and a file named FN, the VALUE option would return different results depending on which type of argument is expected.

alternative

can be any of these:

ASSIGN [/ SYNTAX /]

matches an existing logical-unit name. The name must be terminated by a standard TACL separator. The SYNTAX option specifies that any logical-unit name is acceptable as long as it is formatted correctly.

ATTRIBUTENAME

matches a valid DEFINE attribute name as described in Section 5, Statements and Programs. The name must be terminated by a standard TACL separator.

ATTRIBUTEVALUE

matches a valid DEFINE attribute value as described in Section 5, Statements and Programs. The name must be terminated by a standard TACL separator.

CHARACTERS / *char-option* [, *char-option*] ... /

matches a contiguous sequence of characters starting at an arbitrary position in the argument sequence.

char-option is either of these:

START *num*

specifies the start of the sequence of characters, in the range -1 to +32767. -1 specifies the current position. Positive numbers specify absolute character positions: Zero specifies the first character of the argument set (the space following the routine name); higher numbers specify character positions to the right of that. If you omit this option, the starting position defaults to the current position.

WIDTH *num*

specifies the number of characters to process. If you omit this option, it defaults to one. If the specified number of characters are found, the current position moves to the point just beyond the last character processed. If the portion of the argument beyond the starting point contains fewer characters than specified by WIDTH, an error occurs.

CLOSEPAREN

matches a closing parenthesis.

COMMA

matches a comma.

DEFINENAME

matches a valid DEFINE name as described in Section 5, Statements and Programs. The name must be terminated by a standard TACL separator.

DEFINETEMPLATE

matches a valid DEFINE template as described in Section 5, Statements and Programs. The name must be terminated by a standard TACL separator.

DEVICE [/ SYNTAX /]

matches the name of an existing device. The SYNTAX option specifies that any device name is acceptable as long as it is formatted correctly.

END

matches the closing square bracket, vertical line, or end-of-line (if the routine invocation is not enclosed in square brackets) that marks the end of the argument sequence.

FILENAME [/ *file-option* [, *file-option*] ... /]

matches the name of an existing file or a DEFINE name that references an existing file.

file-option can be either of these:

SEARCHLIST *search-place* [*search-place*] ...

specifies the locations to be searched for the named file.

search-place can be either of these:

subvol

specifies the subvolume to be searched.

#DEFAULTS

specifies that your current default subvolume is to be searched. Do not use square brackets around #DEFAULTS.

SYNTAX

specifies that any file name or DEFINE name is acceptable as long as it is formatted correctly. The file need not exist. If you do not include SYNTAX and you specify a file name, the file name returned by VALUE is fully qualified, using your current defaults for any missing fields.

GMOMJOBID

matches a valid job ancestor job ID, expressed as one of these:

```
$process-name. num
cpu,pin, num
```

such as $XYZ.3 or 2,12,100. num is a positive integer. The value of pin can range from 0 to 65535.

JOBID

matches a valid job ID expressed as a signed integer.

KEYWORD / WORDLIST *keyword* [*keyword*] ... /

matches any one of a specified list of words (WORDLIST is required; it must precede the first keyword in the list).

NUMBER [/ *range-option* [, *range-option*] /]

matches a number, the name of a variable that contains a numeric value, or an arithmetic expression enclosed in parentheses. It also matches an argument that begins with one or more numeric digits (0-9); if the argument also contains nonnumeric data, the current position pointer stops immediately following the last digit. The VALUE option obtains the value of the number.

range-option can be either of these:

MINIMUM *num*

specifies the minimum valid number. num is an integer.

MAXIMUM *num*

specifies the maximum valid number. num is an integer.

OPENPAREN

matches an opening parenthesis.

OTHERWISE

specifies that TACL should not invoke the TACL error handler when #ARGUMENT detects an error (such as an invalid argument). Instead, TACL allows that the program to handle the error. TACL does not move the pointer to the current position argument and does not return the current argument. If you

use OTHERWISE, it must be positioned as the last option in the list of alternatives.

`PARAM [ / SYNTAX / ]`

matches the name of an existing parameter. The SYNTAX option specifies that any parameter name is acceptable as long as it is formatted correctly.

`PARAMVALUE`

matches a text sequence of 1 to 255 characters terminated by a comma, space, or end-of-line.

If text is enclosed in quotation marks, it can contain spaces and commas, and can have leading or trailing spaces. Contained quotation marks must be doubled (""). (The enclosing quotes, and one of each pair of contained quotes, do not count against the 255-character limit.) Use two adjacent quotation marks to express an empty argument.

If the argument sequence is received from the IN file and #INFORMAT is set to TACL, enclosing the text in quotation marks does not override the significance of the TACL special characters [,], |, ==, {, }, and ~. Precede a special character with a tilde to make TACL treat it as text (for example, ~~).

`PROCESSID [ / SYNTAX / ]`

matches the process name or cpu, pin of a process. The SYNTAX option specifies that any process specifier (process name or cpu, pin of a process) is acceptable as long as it is formatted correctly. The value of pin can range from 0 to 65535. If you do not include the SYNTAX option, TACL matches the process name or cpu, pin of an existing process.

`PROCESSNAME [ / SYNTAX / ]`

matches the name of a process. The SYNTAX option specifies that any process name is acceptable as long as it is formatted correctly. If you do not include the SYNTAX option, TACL matches the process name of an existing process.

`SECURITY [ / LENGTH` *`num`* `/ ]`

matches a security string, which must be enclosed in quotation marks. Unless the LENGTH option specifies a different length, the security string must have four characters (the quotation marks are not considered as part of the length).

`SEMICOLON`

matches a semicolon.

`SLASH`

matches a slash.

STRING

matches the name of a variable level, text enclosed in quotation marks, or a concatenation of such entities. The concatenation operator is '+'. With this type of alternative, the TEXT option returns an exact copy of the string as it was entered; the VALUE option returns the resulting value of the string after it has been evaluated (enclosing quotation marks removed and all specified invocation and concatenation performed).

SUBSYSTEM

matches a subsystem ID (for use with #STOP, #ABEND, or the Subsystem Programmable Interface). See the *TACL Programming Guide* for a description of subsystem IDs.

SUBVOL

matches the name of a subvolume.

SUBVOLTEMPLATE

matches a subvolume name that may contain special template characters.

SYSTEMNAME [/ SYNTAX /]

matches the name of a known system. The SYNTAX option specifies that any node name is acceptable as long as it is formatted correctly.

TEMPLATE

matches a file-name template. The template returned by VALUE has defaults substituted for fields omitted in the input.

TEXT

matches all remaining arguments. The text returned by VALUE contains spaces in place of end-of-line delimiters.

TOKEN / TOKEN *token-text* /

matches a user-specified character sequence regardless of delimiters. The TOKEN keyword is required preceding the token text. The token text, 1 to 32 characters in length, is not case-sensitive. It cannot include a comma, left or right parenthesis, semicolon, slash, or space.

TRANSID

matches a TMF transaction ID formatted as follows:

node-name(*crash-count*). *cpu*. *sequence*

If *crash-count* is zero, the transaction ID can be formatted as follows:

node-name. *cpu*. *sequence*

If the transaction ID is local, you can omit the node name.

`USER [ / { SYNTAX | USERNAME } / ]`

matches an existing user name or user ID. The SYNTAX option states that the user name or user ID need not exist but need only be formatted correctly (#USERNAME or #USERID can determine whether it is defined). The USERNAME option specifies that the argument must be a user name only; a user ID is unacceptable.

`VARIABLE [ / var-option [ , var-option ] ... / ]`

matches the name of an existing variable.

var-option can be any of these:

`ALLOW type [ type ... ]`

specifies the type(s) of variables that are acceptable.

`FORBID type [ type ... ]`

specifies the type(s) of variables that are unacceptable.

In both of the preceding options, *type* is one of:

```
ALIAS
DELTA
DIRECTORY
ITEM (a STRUCT item)
MACRO
ROUTINE
STRUCT
TEXT
```

If you use ALLOW, only the types listed are allowed; if you use FORBID, all types except those listed are allowed. If you use neither, all types of variables are acceptable. You cannot specify both ALLOW and FORBID as options to the same VARIABLE type.

`QUALIFIED`

specifies that the variable name must be qualified by a level number.

`SYNTAX`

specifies that any variable name is acceptable as long as it is formatted correctly. If you omit the SYNTAX option, the VALUE result is qualified by an absolute level.

`UNQUALIFIED`

specifies that the variable name must not be qualified by a level number.

If you omit both QUALIFIED and UNQUALIFIED, a variable name is acceptable with or without a level number.

`WORD [ / SPACE / ]`

matches all text up to the next comma, slash, semicolon, space, left or right parenthesis, or end-of-line. If you include the SPACE option, space and end-of-line are the only delimiters.

Result

#ARGUMENT returns the position number of the alternative that matches the type of the routine argument being examined.

Considerations

- #ARGUMENT can appear only in a routine.
- If the routine argument does not match any of the alternatives tested for by #ARGUMENT, the TACL error handler takes over. If you include the OTHERWISE alternative, however, no error can occur; you must construct your own error-checking mechanism to deal with invalid arguments.
- If the call to a routine is issued from the IN file or is loaded under control of a ?FORMAT directive in a library file, the setting of the #INFORMAT built-in variable has an effect on the way arguments are processed. This is summarized in Table 9-3. In each case, the #ARGUMENT function is expecting a STRING argument. For sake of example, the variable SV is presumed to contain the text SUBVOL. and the variable FN contains FILENAME.

Table 9-3. Effect of #INFORMAT on Argument Processing

#INFORMAT	Argument	Text	Value
Plain or quoted	sv'+'"[fn]"	sv'+'"[fn]"	SUBVOL.[fn]
TACL	sv'+'"[fn}"	sv'+'"FILENAME"	SUBVOL.FILENAME

Under control of PLAIN format, TACL does not recognize square brackets as metacharacters, so does not invoke the variable. Under QUOTED format, the quotation marks instruct TACL to treat the square brackets as ordinary text.

For arguments not affected by #INFORMAT, the TEXT and VALUE options behave in the same way as shown for TACL input format.

- The identity of the alternative being tested can have an effect on the result of the VALUE option. For example, suppose that in addition to the variable FN, above, there were a file of the same name. Using a single argument, the VALUE option could return different results depending on which type of argument is expected; examples of this are shown in Table 9-4 on page 9-29.

Table 9-4. Some Effects of Expectation on VALUE Result

Alternative	Argument	Text	Value
String	fn	fn	FILENAME
Variable	fn	fn	:MYDIR:FN.1
Filename	fn	fn	\SYS.$VOL.SUBVOL.FN

Other special VALUE results are documented in the descriptions of the individual alternatives. In the majority of cases, however, TEXT and VALUE results are the same.

- As shown in the preceding table, a given argument can meet the criteria for more than one alternative. When this occurs, TACL uses the first such alternative tested and places its interpretation (if any) on the VALUE result. You should, therefore, order the alternatives in an #ARGUMENT call so that the alternative that corresponds to the type of argument most likely to be processed by #ARGUMENT appears first in the alternatives list.

Examples

1. This example shows the use of #ARGUMENT to examine the argument list of a routine.

```
#PUSH arg
[#LOOP |DO|
  [#CASE [#ARGUMENT /VALUE arg/ FILENAME SUBVOL END]
    | 1 | #OUTPUT Argument is existing file named [arg].
    | 2 | #OUTPUT Argument is valid subvolume [arg].
    | 3 | #OUTPUT No more arguments to parse.
  ] == End of case
|UNTIL| NOT [#MORE]
] == End of loop
```

2. This example shows how a STRING argument can be processed. Assume that RTN is a routine defined as follows:

```
#FRAME
#PUSH txt vlu

== Get a string, using #IF to discard #ARGUMENT result
#IF [#ARGUMENT /TEXT txt, VALUE vlu/ STRING]

== Get end-of-args, using #IF to discard #ARGUMENT result
#IF [#ARGUMENT END]

== Show string as originally entered
#OUTPUTV "TEXT: " '+' txt

== Show string after evaluation and concatenation
#OUTPUTV "VALUE: " '+' vlu
#UNFRAME
```

Assume also that the variable TERMNAME contains the name of the home terminal, which currently is $MINE, and that RTN is invoked as follows:

```
RTN "My terminal is " '+' termname '+' " at this time."
```

The resulting output is:

```
TEXT: "My terminal is " '+' termname '+' " at this time."
VALUE: My terminal is $MINE at this time.
```

For additional examples, see the *TACL Programming Guide*.

#ASSIGN Built-In Variable

#ASSIGN contains all the defined attributes of all currently defined unit names.

```
#ASSIGN [ / option [ , option ] ... / logical-unit ]
```

option

specifies one of these types of information:

ACCESS

requests the type of access for the logical unit; it returns I/O, INPUT, OUTPUT, or nothing.

BLOCK

requests the block size of the logical unit; it returns the block size or nothing.

CODE

requests the file code for the logical unit; it returns the file code or nothing. (See the FUP INFO command in the *File Utility Program (FUP) Reference Manual*.

EXCLUSION

requests the type of exclusion assigned to the logical unit; it returns EXCLUSIVE, PROTECTED, SHARED, or nothing.

EXISTENCE

returns -1 if the logical unit is defined; if not, it returns 0.

FILENAME

requests the file name of the logical unit; it returns the name of the file or nothing.

PRIMARY

requests the primary-extent size for the logical unit; it returns the primary-extent size or nothing.

RECORD

requests the record size for the logical unit; it returns the record size or nothing.

SECONDARY

requests the secondary-extent size of the logical unit; it returns the secondary-extent size or nothing.

specifies the logical-unit name about which you want information. For more information about logical units, see the ASSIGN Command on page 8-21.

Result

#ASSIGN returns as its result a space-separated list of the requested information about the specified logical unit. If you do not specify any options, #ASSIGN returns a space-separated list of all currently defined logical units.

Considerations

- When you first log on, #ASSIGN is initialized to a null value.
- When a backup TACL process takes over, TACL deletes existing assignments.
- All options but EXISTENCE are likely to return nothing if the logical unit is currently undefined, or if it is defined but the particular information was omitted when it was defined.
- It is best not to include multiple options as it is difficult to match results with options when some results are null.
- Use #PUSH #ASSIGN (or PUSH #ASSIGN) to save a copy of all your assigns.
- Use #POP #ASSIGN (or POP #ASSIGN) to replace all your current assigns with those last pushed.
- Use #SET #ASSIGN (or SET VARIABLE #ASSIGN) to set a logical unit either with or without definitions:
  - If you use #SET #ASSIGN without any arguments, all logical units become undefined.
  - If you supply only a logical-unit name, that logical unit becomes undefined.
  - When you supply both a logical-unit name and options, the logical unit definition is set according to the options; any previous definition of the logical unit is lost.
  - Options are processed in the order supplied.
  - If options conflict, the last one to appear is used.

The syntax for #SET #ASSIGN is:

```
#SET #ASSIGN [ [ / option [ , option ] ... / ] logical-unit ]
```

option

specifies the information that is to be defined for the logical unit; it can be any of these:

`ACCESS { I/O | INPUT | OUTPUT }`

specifies the type of access for the logical unit.

`BLOCK` *num*

specifies the block size of the logical unit.

`CODE` *num*

specifies the file code for the logical unit.

`EXCLUSION { EXCLUSIVE | PROTECTED | SHARED }`

specifies the type of exclusion assigned to the logical unit.

`EXISTENCE`

specifies that a logical unit can be defined without including any other information.

`FILENAME` *file-name*

specifies the file name of the logical unit.

`PRIMARY` *num*

specifies the primary-extent size for the logical unit being assigned.

`RECORD` *num*

specifies the record size for the logical unit.

`SECONDARY` *num*

specifies the secondary-extent size of the logical unit.

logical-unit

specifies the logical-unit name to be defined with the indicated options.

#BACKUPCPU Built-In Function

Use #BACKUPCPU to start a backup process for the current TACL process or delete an existing backup process.

```
#BACKUPCPU [ cpu ]
```

cpu

specifies the number, in the range 0 to 15, of the CPU to be used for the backup. If you omit it, TACL runs without a backup.

Result

#BACKUPCPU returns nothing.

Considerations

- #BACKUPCPU with no CPU specification has no effect if the current TACL process has no backup. If there is a backup, TACL deletes it.
- Your TACL must be a named process to have a backup.
- The specified CPU module must exist on your system.
- The backup CPU cannot be the same as the primary CPU of your TACL.
- The backup CPU need not be running at the time that #BACKUPCPU is issued.
- If you specify a CPU for a backup process and a backup process already exists in any CPU, an error message appears.
- TACL switches to the backup process if the primary CPU becomes unavailable. After the primary CPU is reloaded, TACL switches back to the primary CPU unless a user is logged on, in which case TACL postpones switching until the user logs off.
- To force TACL to switch to the backup process, see the #SWITCH Built-In Function on page 9-395.
- For examples, see the BACKUPCPU Command on page 8-28.

#BEGINTRANSACTION Built-In Function

Use #BEGINTRANSACTION to start a new transaction. This function invokes the BEGINTRANSACTION operating system procedure. When you use this function, the TMF subsystem creates a new transaction identifier that becomes the current transaction identifier for the process issuing BEGINTRANSACTION.

```
#BEGINTRANSACTION
```

Result

#BEGINTRANSACTION returns zero if it is successful, or a file-system error indicating the reason it failed.

#BREAKMODE Built-In Variable

Use #BREAKMODE to disable, enable, or temporarily postpone most of the effect of the BREAK key.

```
#BREAKMODE
```

Result

#BREAKMODE returns the current break mode setting: DISABLE, ENABLE, or POSTPONE.

Considerations

- Use #PUSH #BREAKMODE (or PUSH #BREAKMODE) to save the current break mode setting.
- Use #SET #BREAKMODE (or SET VARIABLE #BREAKMODE) to establish a new break mode setting.
- The syntax for #SET #BREAKMODE is:

```
#SET #BREAKMODE { DISABLE | ENABLE | POSTPONE }
```

`DISABLE`

immediately turns off the BREAK key. Pressing the key has no further effect, nor is control of the key given to another process. If the previous break mode was POSTPONE and someone had pressed the BREAK key, TACL discards the postponed break.

`ENABLE`

immediately turns on the BREAK key; TACL takes control of the key without noting whether control had been taken by another process. If the previous break mode was POSTPONE and someone had pressed the BREAK key, the postponed break immediately takes effect as though the key had just been pressed.

`POSTPONE`

turns on the BREAK key until someone presses the key once. After the key is pressed, TACL turns off the BREAK key; further breaks have no effect. Control of the BREAK key is not given to another process. If the previous break mode was POSTPONE and someone had pressed the BREAK key, TACL discards the postponed break. If there is active I/O on IN or OUT when BREAK is pressed the final time before it is turned off, that I/O operation is retried.

- Use #POP #BREAKMODE (or POP #BREAKMODE) to restore the break mode to its previous setting.
- When you first log on, #BREAKMODE is initialized to ENABLE.

#BREAKPOINT Built-In Function

Use #BREAKPOINT to set or delete a _DEBUGGER breakpoint for a specific variable level.

```
#BREAKPOINT variable-level state
```

variable-level

is the name of an existing variable level.

state

is either a nonzero value, which sets a breakpoint for the variable level, or zero, which deletes an existing breakpoint.

Result

#BREAKPOINT returns the previous breakpoint state of the variable level: 0 if there was no breakpoint, -1 if there was one.

Considerations

- A debugging breakpoint causes TACL to invoke _DEBUGGER just before invoking a variable level having the breakpoint. Only invocation has this effect; no other usage of the variable level (pushing, setting, and so on) causes TACL to invoke _DEBUGGER.
- For additional information about the TACL debugger, see the DEBUGGER Function on page 8-51.

#BUILTINS Built-In Function

Use #BUILTINS to obtain the names of the TACL built-in functions or built-in variables.

```
#BUILTINS [ / { FUNCTIONS | VARIABLES } / ]
```

FUNCTIONS

displays the built-in functions.

VARIABLES

displays the built-in variables.

If you fail to specify either option, FUNCTIONS is assumed.

Result

#BUILTINS returns a space-separated list of the names of the specified built-in functions or variables.

#CASE Built-In Function

Use #CASE to return one of a set of alternative text sequences. You provide a set of labeled options using an enclosure. When TACL evaluates the #CASE statement, it searches the enclosure for a label that matches some specified text, and returns the text sequence that follows the label.

```
#CASE text enclosure
```

text

is the text to be matched to a label in the enclosure; it cannot contain any spaces.

enclosure

is an enclosure consisting of a series of labels and options as follows:

```
| label-1 | [ option-text-1 ]
| label-2 | [ option-text-2 ]
...
| OTHERWISE | [ option-text-n ]
```

The OTHERWISE label, if included, must be listed last. For more information about enclosures, see Section 5, Statements and Programs.

label-num

is a unique label that identifies an option. TACL searches the labels for one matching the text. All labels must be outside of any square brackets within enclosure and between vertical bars. A label can be a space-separated list, allowing several values of text to match a single label.

option-text-num

is a text sequence, typically one or more functions that can be executed when returned by #CASE.

Result

#CASE returns all option text that follows a selected label; the text is terminated by the next label or the end of the enclosure. The result includes any spaces or carriage returns that precede the text. If text does not match any label, #CASE returns the text following the OTHERWISE label, up to the next label or to the end of the enclosure.

Considerations

- No label can appear more than once in the enclosure.
- Label evaluation is not case-sensitive. #CASE treats a label of the form | A a | as a repetition of the same label, resulting in an error.

- You cannot use square brackets in a *label-num*; no evaluation takes place between the vertical bars.
- If neither a matching *label-num* nor OTHERWISE is present, an error occurs.

Examples

1. This macro accepts an argument. If the argument is 0, the macro writes Zero; if the argument is 1, the macro writes One, and so on.

```
[#CASE %1%
  | 0 | #OUTPUT Zero
  | 1 | #OUTPUT One
  | 2 | #OUTPUT Two
  | 3 | #OUTPUT Three
  | 4 | #OUTPUT Four
  | 5 | #OUTPUT Five
  | 6 | #OUTPUT Six
  | 7 | #OUTPUT Seven
  | 8 | #OUTPUT Eight
  | 9 | #OUTPUT Nine
  | OTHERWISE | #OUTPUT %1% is not a digit.
]
```

2. This example illustrates the use of a space-separated list as a label:

```
[#CASE %1%
  | 0 2 4 6 8 | #OUTPUT Even
  | 1 3 5 7 9 | #OUTPUT Odd
  | OTHERWISE | #OUTPUT Isn't a digit
]
```

3. This example shows how multiple lines can be used as option text for a label:

```
[#CASE %1%
  | 1 3 5 7 9 | #OUTPUT Odd
                #OUTPUT Prime
  | 0 2 4 6 8 | #OUTPUT Even
  [#IF [#MATCH %1% 2] |THEN|
                #OUTPUT Prime]
  |OTHERWISE|   #OUTPUT Not a digit
]
```

#CHANGEUSER Built-In Function

Use #CHANGEUSER to log on under a different user ID, keeping the current user's values.

Note. The LOGON command behavior depends on the Safeguard environment. If Safeguard is not running on your system or if the USER_AUTHENTICATE procedure is not in the system library, the alias option is not available

---

```
#CHANGEUSER [ / CHANGEDEFAULTS / ]
group-name.user-name |  group-id,user-id | alias} password
```

CHANGEDEFAULTS

causes TACL to set its default node, volume, subvolume, and file security to that of the new user instead of retaining the current user's values.

group-name. *user-name* or *group-id*, *user-id*

is the group name and user name, or the group number and user number, of the user who is logging on. The user identity must already have been established. If you do not have a user account, see your system administrator.

If the TACL configuration NAMELOGON option is not set to 0, or if Safeguard is running and the Safeguard NAMELOGON flag is not set to 0 (for the USER_AUTHENTICATE_call), the group number and user number is not accepted.

alias

is an alternate assigned name. Each alias must be unique within the local system. An alias is a case-sensitive text string that can be up to 32 alphanumeric characters in length. In addition to alphabetic and numeric characters, the characters period (.), hyphen (-), and underscore(_) are permitted within the text string. The first character of an alias must be alphabetic or numeric. For more information on aliases, see the *Safeguard Reference Manual*.

password

is the password associated with the user.

Result

#CHANGEUSER returns -1 if it is successful; if not, it returns 0.

Considerations

- If the USER_AUTHENTICATE_procedure exists in the system library, TACL calls the USER_AUTHENTICATE_ procedure. USER_AUTHENTICATE_ uses the

Safeguard facility if Safeguard is running. Otherwise, the VERIFYUSER system procedure is called by TACL.

- If the user ID and password are syntactically correct (but not necessarily valid), TACL sends a pre-LOGON message to $CMON (if that user-supplied monitoring process exists) for additional validation before calling VERIFYUSER. In addition, TACL sends LOGON messages to $CMON during the logon process. For more information about $CMON, see Section 6, The TACL Environment.
- If the logon operation is rejected, either by the USER_AUTHENTICATE_ procedure or by $CMON, TACL notifies you of the failure but does not specify whether user-name or password was wrong.
- If the USER_AUTHENTICATE_ procedure fails to recognize the user information given during the logon operation, the TACL built-in variable #ERRORNUMBERS contains the error information. To display the error information, issue these commands:

```
#PUSH n1 n2 n3 n4
#SETMANY n1 n2 n3 n4, [#ERRORNUMBERS]
```

where:

```
n1 = 1074 (Invalid user name or password)
n2 = error return from USER_AUTHENTICATE_
n3 = error return detail from USER_AUTHENTICATE_
n4 = 0
```

If n2 contains 0 (no error state returned), USER_AUTHENTICATE_ is not in the system library or Safeguard is not running. If Safeguard is not running, n3 and n4 are set to 0.

- TACL sends an “illegal LOGON” message (code -53) to $CMON on the third and all subsequent logon failures until a logon succeeds. Each time it sends the “illegal LOGON” message, TACL displays the $CMON reply (if not empty). Safeguard and USER_AUTHENTICATE_ security logic can impose a delay before the next prompt. You cannot exit from the delay by pressing the BREAK key.
- Both the LOGON command (issued while you are already logged on) and the #CHANGEUSER built-in function change your identity while keeping your previous IDs, variables, segment files, and so on. The principal difference is that with LOGON you assume both the saved (logon) default subvolume and the current default subvolume of the new ID; with #CHANGEUSER, you assume the saved default subvolume of the new ID, but remain in the subvolume that was current at the time you invoked the function.
- You can configure TACL to disable logons from a logged-on state. To alter the configuration, CMON should reply with the NOCHANGEUSER field set to -1. For more information, see the *Guardian Programmer’s Guide*. If the capability is disabled, #CHANGEUSER fails and returns a zero. To display the value of NOCHANGEUSER, use #GETCONFIGURATION.

- The TACL configuration BLINDLOGON option and the Safeguard BLINDLOGON flag do not affect #CHANGEUSER.
- If the TACL configuration option REMOTECMONREQUIRED is not set to 0, all operation requiring approval by remote $CMON are denied if that remote $CMON is unavailable or is running too slowly. To display the value of REMOTECMONREQUIRED, use #GETCONFIGURATION /REMOTECMONREQUIRED/.

#CHARACTERRULES Built-In Variable

Use #CHARACTERRULES to obtain or set the name of the file that is currently being used to define the character-processing rules.

```
#CHARACTERRULES
```

Result

#CHARACTERRULES returns the fully qualified name of the file containing the character-processing rules currently in effect. This name is CPRULES0, CPRULES1, or the name of a user-supplied file that contains character rules.

Considerations

- When starting a TACL process, if #CHARACTERRULES is empty after TACLLOCL has been invoked, TACL selects CPRULES0. TACL searches for CPRULES0 in $SYSTEM.SYSTEM, and if CPRULES is not found there, then TACL searches the same volume and subvolume in which the TACL program file resides. TACL reports a warning if a CPRULES file does not exist when the user logs on and TACL tries to access a CPRULES file.
- Use #PUSH #CHARACTERRULES or (PUSH #CHARACTERRULES) to save a copy of the current character-processing file name and processing rules. The rules currently in effect are unchanged. Any error that occurs while pushing #CHARACTERRULES causes the variable to be popped at once. The current rules remain unchanged.
- Use #POP #CHARACTERRULES or (POP #CHARACTERRULES) to revert to the previous character-processing rules file name and its rules. TACL does not reopen the file and reread the rules; instead, it uses the copy of the rules that it saved during the previous #PUSH operation.
- Use #SET #CHARACTERRULES (or SET VARIABLE #CHARACTERRULES) to define the name of the file to use as the character-processing rules file.

The syntax of #SET #CHARACTERRULES is:

```
#SET #CHARACTERRULES file-name
```

file-name

is the name of a file containing character-processing rules. TACL searches for the file in the subvolumes specified in #PMSEARCHLIST, opens the file, reads in the character-processing rules, and closes the file. These conditions also apply:

- TACL maintains an internal copy of the rules.

- If an error occurs while acquiring the new rules, the character rules already in effect remain unchanged.
- If a #SET #CHARACTERRULES is never executed, TACL uses default rules.

#CHARADDR Built-In Function

Use #CHARADDR to convert a line address to a character address.

```
#CHARADDR variable-level line-addr
```

variable-level

is an existing variable level to be operated upon. It must not be a DIRECTORY, a STRUCT, or a STRUCT item.

line-addr

is an integer or a variable level that contains an integer. *line-addr* specifies the line number of the line to be converted. The line number must be in the range from 1 to *max-int*, inclusive.

Result

#CHARADDR returns the address of the first character of the specified line.

Considerations

- If *line-addr* is greater than the number of lines in the variable, #CHARADDR returns the character address of the last end-of-line, plus one.
- Each logical line contains an end-of-line character that counts as one byte. For variables that contain TACL statements, each metacharacter uses the number of visible characters plus one, including unprintable characters that are subject to change from one TACL RVU to another. Nonmetacharacters use one byte.
- If *variable-level* is empty, #CHARADDR returns zero.

Example

Assume that var is a variable level containing these characters:

```
ABCDEFG
HIJKLMNOPQRST
UVWXYZ
```

The invocation:

```
#CHARADDR var 2
```

returns 9; the letter H at the beginning of line 2 is the ninth character in var, counting the end-of-line between lines 1 and 2 as one character.

The invocation:

```
#CHARADDR var 100
```

returns 30: the 26 characters, three end-of-line characters, plus one.

#CHARBREAK Built-In Function

Use #CHARBREAK to insert an end-of-line into a variable level at a specified character position.

```
#CHARBREAK variable-level char-addr
```

variable-level

is the name of an existing variable level into which the end-of-line is to be inserted. It must not be in a shared segment and must not be a DIRECTORY, a STRUCT, or a STRUCT item.

char-addr

is the character position at which the end-of-line is to be inserted.

Result

#CHARBREAK returns nothing.

Considerations

- Each line break contains an internal end-of-line character that counts as one byte.
- #CHARBREAK inserts the end-of-line immediately preceding the existing character at *char-addr*.
- The end-of-line is inserted regardless of whether there is already an end-of-line at *char-addr*.
- If *char-addr* is 1, an end-of-line is inserted at the beginning of the variable level.
- If *char-addr* is beyond the end of the variable level, no end-of-line is inserted.
- IF *char-addr* is less than 1, an error occurs.

Example

Assume that var is a variable level containing:

```
ABCDEFG
HIJKLMNOPQRST
UVWXYZ
```

The invocation:

```
#CHARBREAK var 13
```

causes var to contain:

```
ABCDEFG
HIJK
```

```
LMNOPQRST
UVWXYZ
```

#CHARCOUNT Built-In Function

Use #CHARCOUNT to obtain the number of characters in a variable level.

```
#CHARCOUNT variable-level
```

variable-level

is the name of an existing variable level whose characters are to be counted. It must not be a DIRECTORY, a STRUCT, or a STRUCT item.

Result

#CHARCOUNT returns the number of characters in the variable level, counting each end-of-line except the last as a character and counting each internal representation of [, |, or] as multiple characters.

Considerations

- To make a quoted string yield the same character count as a variable level containing the same text (minus the quotation marks), the final end-of-line in a variable level is not counted. This means that #CHARCOUNT does not necessarily yield the same result as #VARIABLEINFO /OCCURS/, which includes all end-of-lines in its count.
- Each logical line contains an end-of-line character that counts as one byte. For variables that contain TACL statements, each metacharacter uses the number of visible characters plus one, including unprintable characters that are subject to change from one TACL RVU to another. Nonmetacharacters use one byte.

Examples

1. Assume that var is a variable level containing:

   ```
   ABCDEFG
   ```

   The invocation:

   ```
   #CHARCOUNT var
   ```

   returns 7; #CHARCOUNT does not count the last end-of-line character in a variable level as a character.

2. Assume that var is a variable level containing:

   ```
   ABCDEFG
   HIJKLMNOPQRST
   UVWXYZ
   ```

   The invocation:

   ```
   #CHARCOUNT var
   ```

returns 28; there are 26 letters and 2 internal line breaks. #CHARCOUNT does not count the last end-of-line character.

#CHARDEL Built-In Function

Use #CHARDEL to delete characters from a variable level, starting at a character address.

```
#CHARDEL variable-level char-addr-1
  [ [ FOR char-count ] | [ TO char-addr-2 ] ]
```

variable-level

is an existing variable level from which the characters are to be deleted. It must not be in a shared segment and must not be a DIRECTORY, a STRUCT, or a STRUCT item.

char-addr-1

is an integer greater than zero or a variable level that contains an integer greater than zero. *char-addr-1* specifies the character position at which character deletion is to begin. The character position must be in the range from 0 to *max-int*, inclusive.

char-count

is an integer or a variable level that contains an integer. *char-count* specifies the number of lines to be deleted. The line count must be in the range from 0 to *max-int*, inclusive.

char-addr-2

is an integer or a variable level that contains an integer. *char-addr-2* specifies the line number at which line deletion is to end. The line address must be in the range from 0 to *max-int*, inclusive.

Result

#CHARDEL returns nothing.

Considerations

- If you use TO, the character specified by *char-addr-2* is included in the deletion. That is, “x TO y” is equivalent to “x FOR (y-x+1).”
- If you use TO and *char-addr-1* is greater than *char-addr-2*, or if you use FOR and *char-count* is zero, no deletion occurs.
- If you omit both FOR and TO, TACL deletes the character specified by *char-addr-1*.
- Any part of the specified deletion that lies beyond the end of the variable level is ignored.

- Each logical line contains an end-of-line character that counts as one byte. For variables that contain TACL statements, each metacharacter uses the number of visible characters plus one, including unprintable characters that are subject to change from one TACL RVU to another. Nonmetacharacters use one byte.

Examples

Assume that var is a variable level containing the following, including one internal end-of-line character after each line:

```
ABCDEFG
HIJKLMNOPQRST
UVWXYZ
```

1. Either of the invocations:

   ```
   #CHARDEL var 7 TO 9 or #CHARDEL var 7 FOR 3
   ```

   causes var to contain:

   ```
   ABCDEFIJKLMNOPQRST
   UVWXYZ
   ```

2. Either of the invocations:

   ```
   #CHARDEL var 7 TO 100 or #CHARDEL var 7 FOR 94
   ```

   causes var to contain:

   ```
   ABCDEF
   ```

#CHARFIND Built-In Function

Use #CHARFIND to find text in a variable level, searching forward from a specified character address.

```
#CHARFIND [ / EXACT / ] variable-level char-addr text
```

EXACT

specifies that the search is to be case-sensitive; if you omit it, the search makes no distinction between uppercase and lowercase letters.

variable-level

is an existing variable level within which TACL will search for text. It must not be a DIRECTORY, a STRUCT, or a STRUCT item.

char-addr

is an integer greater than zero or a variable level that contains an integer greater than zero. *char-addr* specifies the character address at which the search is to begin. The character address must be in the range from 1 to *max-int*, inclusive.

text

is the text constant to be found. The largest valid text length is 32,000 words minus the current contents of the stack. The amount of remaining space is typically 25,000 words long.

Result

#CHARFIND returns the character address at which text begins. If text is not found, #CHARFIND returns zero.

Considerations

- If *char-addr* is past the end of the variable level, #CHARFIND returns zero.
- A text specification can include internal end-of-line characters if the entire invocation is enclosed in square brackets, but leading and trailing end-of-lines and spaces are ignored.
- The search begins immediately at the character address specified. If you make repeated calls to this function, using the result of each as a starting point for the next, you must add one to that result before supplying it to a subsequent call.
- If *variable-level* is empty, #CHARFIND returns zero.
- Each logical line contains an end-of-line character that counts as one byte. For variables that contain TACL statements, each metacharacter uses the number of

visible characters plus one, including unprintable characters that are subject to change from one TACL RVU to another. Nonmetacharacters use one byte.

Examples

Assume that var is a variable level containing:

```
ABCDEFG
HIJKLMNOPQRST
UVWXYZ
```

1. The invocation:

   ```
   #CHARFIND var 1 IJK
   ```

   returns 10; the first occurrence of IJK starting at or after 1 is at character address 10.

2. The invocation:

   ```
   #CHARFIND var 10 IJK
   ```

   returns 10; the first occurrence of IJK is exactly at the starting character address, 10.

3. The invocation:

   ```
   #CHARFIND var 11 IJK
   ```

   returns 0; there are no occurrences of IJK starting at or after character address 11.

4. The invocation:

   ```
   #CHARFIND var 1 FOO
   ```

   returns 0; there are no occurrences of FOO anywhere in var.

#CHARFINDR Built-In Function

Use #CHARFINDR to find text in a variable level, searching backward from a character address.

```
#CHARFINDR [ / EXACT / ] variable-level char-addr text
```

EXACT

specifies that the search is to be case-sensitive; if you omit it, the search makes no distinction between uppercase and lowercase letters.

variable-level

is an existing variable level within which TACL will search for text. It must not be a DIRECTORY, a STRUCT, or a STRUCT item.

char-addr

is an integer greater than zero or a variable level that contains an integer greater than zero. *char-addr* specifies the character address at which the search is to begin. The character address must be in the range from 1 to *max-int*, inclusive. The search moves backward from this point.

text

is the text constant to be found. The largest valid text length is 32,000 words minus the current contents of the stack. The amount of remaining space is typically 25,000 words long.

Result

#CHARFINDR returns the character address at which text begins. If text is not found, #CHARFINDR returns zero.

Considerations

- If *char-addr* is past the end of the variable level, #CHARFINDR starts the search at the end of the contents of the variable.
- A text specification can include internal end-of-line characters if the entire invocation is enclosed in square brackets, but leading and trailing end-of-lines and spaces are ignored.
- The search begins immediately at the character address specified. Because the search does not find a match unless the entire matching text appears at or before *char-addr*, you must specify a starting address at the end of a variable level (#CHARCOUNT or greater) to find text at the end of it.
- If *variable-level* is empty, then #CHARFINDR returns zero.

- Each logical line contains an end-of-line character that counts as one byte. For variables that contain TACL statements, each metacharacter uses the number of visible characters plus one, including unprintable characters that are subject to change from one TACL RVU to another. Nonmetacharacters use one byte.

Examples

Assume that var is a variable level containing:

```
ABCDEFG
HIJKLMNOPQRST
UVWXYZ
```

1. The invocation:

   ```
   #CHARFINDR var 28 IJK
   ```

   returns 10; the nearest occurrence of IJK ending before character address 28 starts at character address 10.

2. The invocation:

   ```
   #CHARFINDR var 14 IJK
   ```

   returns 10; the nearest occurrence of IJK ending before character address 14 starts at character address 10.

3. The invocation:

   ```
   #CHARFINDR var 12 IJK
   ```

   returns 10.

4. The invocation:

   ```
   #CHARFINDR var 28 FOO
   ```

   returns 0; there are no occurrences of FOO anywhere in var.

5. This set of statements returns 3:

   ```
   #PUSH x
   #SET x ABCDEFG
   #CHARFINDR x [#CHARCOUNT x] C
   ```

   The occurrence of C is three characters from the start of the contents of variable x.

#CHARFINDRV Built-In Function

Use #CHARFINDRV to find a string constant in a variable level, searching backward from a specified character address.

```
#CHARFINDRV [ / EXACT / ] variable-level char-addr string
```

EXACT

specifies that the search is to be case-sensitive; if you omit it, the search makes no distinction between uppercase and lowercase letters.

variable-level

is an existing variable level within which TACL will search for string. It must not be a DIRECTORY, a STRUCT, or a STRUCT item.

char-addr

is an integer greater than zero or a variable level that contains an integer greater than zero. *char-addr* specifies the character address at which the search is to begin. The character address must be in the range from 1 to *max-int*, inclusive. The search moves backward from this point.

string

is the string constant or the name of a variable level that contains text. string specifies the characters to be found. It must not be in a shared segment, or be a DIRECTORY, a STRUCT, or a STRUCT item. The largest valid string length is 32,000 words minus the current contents of the stack. The amount of remaining space is typically 25,000 words long.

Result

#CHARFINDRV returns the character address at which string begins. If string is not found, #CHARFINDRV returns zero.

Considerations

- If *char-addr* is past the end of the variable level, #CHARFINDRV starts the search at the end of the contents of the variable.
- The trailing end-of-line in string is ignored. Leading and trailing spaces are preserved, as are all other end-of-lines.
- The search begins immediately at the character address specified. Because the search does not find a match unless the entire matching text appears at or before *char-addr*, you must specify a starting address beyond the end of a variable level ([#CHARCOUNT] or greater) to find text at the end of it.
- If *variable-level* is empty, then #CHARFINDRV returns zero.

- Each logical line contains an end-of-line character that counts as one byte. For variables that contain TACL statements, each metacharacter uses the number of visible characters plus one, including unprintable characters that are subject to change from one TACL RVU to another. Nonmetacharacters use one byte.

Examples

Assume that var is a variable level containing:

```
ABCDEFG
HIJKLMNOPQRST
UVWXYZ
```

and that var2 is a variable level containing:

```
IJK
```

1. Either of the invocations:

   ```
   #CHARFINDRV var 28 "IJK"
   ```
   or
   ```
   #CHARFINDRV var 28 var2
   ```

   returns 10; the nearest occurrence of IJK ending before character address 28 starts at character address 10.

2. Either of the invocations:

   ```
   #CHARFINDRV var 11 "IJK"
   ```
   or
   ```
   #CHARFINDRV var 11 var2
   ```

   returns 0; although an occurrence of IJK starts at character address 10, it does not end before or at character address 12.

3. The invocation:

   ```
   #CHARFINDRV var 28 "FOO"
   ```

   returns 0; there are no occurrences of FOO anywhere in var.

#CHARFINDV Built-In Function

Use #CHARFINDV to find a string constant in a variable level, searching forward from a specified character address.

```
#CHARFINDV [ / EXACT / ] string-1 char-addr string-2
```

EXACT

specifies that the search is to be case-sensitive; if you omit it, the search makes no distinction between uppercase and lowercase letters.

string-1

is a string constant or an existing variable level in which TACL will search for *string-2*. It must not be in a shared segment or be a DIRECTORY, a STRUCT, or a STRUCT item. The largest valid string length is 32,000 words minus the current contents of the stack. The amount of remaining space is typically 25,000 words long.

char-addr

is an integer greater than zero or a variable level that contains an integer greater than zero. *char-addr* specifies the character address at which the search is to begin. The character address must be in the range from 1 to *max-int*, inclusive.

string-2

is the string constant or the name of a variable level that contains text. *string-2* specifies the characters to be found. It must not be in a shared segment or be a DIRECTORY, a STRUCT, or a STRUCT item. The largest valid string length is 32,000 words minus the current contents of the stack. The amount of remaining space is typically 25,000 words long.

Result

#CHARFINDV returns the character address at which *string-2* begins. If *string-2* is not found, #CHARFINDV returns zero.

Considerations

- If *char-addr* is past the end of the variable level, #CHARFINDV returns zero.
- The search begins immediately at the character address specified. If you make repeated calls to this function, using the result of each as a starting point for the next, you must add one to that result before supplying it to a subsequent call.
- If *string-1* is empty, #CHARFINDV returns zero.
- Each logical line contains an end-of-line character that counts as one byte. For variables that contain TACL statements, each metacharacter uses the number of

visible characters plus one, including unprintable characters that are subject to change from one TACL RVU to another. Nonmetacharacters use one byte.

Examples

Assume that var is a variable level containing:

```
ABCDEFG
HIJKLMNOPQRST
UVWXYZ
```

and that var2 is a variable level containing:

```
IJK
```

1. Either of the invocations:

   `#CHARFINDV var 1 "IJK"` or `#CHARFINDV var 1 var2`

   returns 10; the first occurrence of IJK starting at or after 1 is at character address 10.

2. Either of the invocations:

   `#CHARFINDV var 10 "IJK"` or `#CHARFINDV var 10 var2`

   returns 10; the first occurrence of IJK is exactly at the starting character address, 10.

3. Either of the invocations:

   `#CHARFINDV var 11 "IJK"` or `#CHARFINDV var 11 var2`

   returns 0; there are no occurrences of IJK starting at or after character address 11.

4. The invocation:

   `#CHARFINDV var 1 "FOO"`

   returns 0; there are no occurrences of FOO anywhere in var.

#CHARGET Built-In Function

Use #CHARGET to obtain a copy of a set of contiguous characters in a variable level.

```
#CHARGET variable-level char-addr-1
  [ [ FOR char-count ] | [ TO char-addr-2 ] ]
```

variable-level

is an existing variable level from which characters are to be copied. It must not be a DIRECTORY, a STRUCT, or a STRUCT item.

char-addr-1

is an integer greater than zero or a variable level that contains an integer greater than zero. *char-addr-1* specifies the character position at which copying is to begin. The character position must be in the range from 1 to *max-int*, inclusive.

char-count

is an integer greater than zero or a variable level that contains an integer greater than zero. *char-count* specifies the number of characters to be copied. The character count must be in the range from 0 to *max-int*, inclusive.

char-addr-2

is an integer greater than zero or a variable level that contains an integer greater than zero. *char-addr-2* specifies the character position at which copying is to end. The character position must be in the range from 1 to *max-int*, inclusive.

Result

#CHARGET returns the copied characters.

Considerations

- If you use TO, the character specified by *char-addr-2* is included in the copy: That is, “x TO y” is equivalent to “x FOR (y-x)+1.”
- If you use TO and *char-addr-1* is greater than or equal to *char-addr-2*, or if you use FOR and *char-count* is less than one, no copying occurs.
- If you omit both FOR and TO, one character is copied.
- If *char-addr-1* is less than 1, an error occurs.
- If any part of the specified copy lies beyond the end of the variable level, that part is ignored.
- If the result of #CHARGET might possibly include one or more internal end-of-lines, you must enclose in square brackets the invocation of the function that obtains that result.

- Each logical line contains an end-of-line character that counts as one byte. For variables that contain TACL statements, each metacharacter uses the number of visible characters plus one, including unprintable characters that are subject to change from one TACL RVU to another. Nonmetacharacters use one byte.

Examples

Assume that var is a variable level containing:

```
ABCDEFG
HIJKLMNOPQRST
UVWXYZ
```

1. Either of the invocations:

   ```
   #CHARGET var 2 TO 4 or #CHARGET var 2 FOR 3
   ```

   returns:

   ```
   BCD
   ```

2. The invocation:

   ```
   #CHARGET var 3 FOR 10
   ```

   returns:

   ```
   CDEFG
   HIJK
   ```

   and therefore the function invocation producing that result must be enclosed in square brackets:

   ```
   [#OUTPUT [#CHARGET var 3 FOR 10]]
   ```

#CHARGETV Built-In Function

Use #CHARGETV to copy a set of contiguous characters from one variable level to another.

```
#CHARGETV var-1 var-2 char-addr-1
  [ [ FOR char-count ] | [ TO char-addr-2 ] ]
```

var-1

is an existing variable level from which characters are to be copied. It must not be a DIRECTORY, a STRUCT, or a STRUCT item.

var-2

is an existing variable level that is to receive the copy. Its original contents are lost and its type is set to TEXT.

char-addr-1

is an integer greater than zero or a variable level that contains an integer greater than zero. *char-addr-1* specifies the character position at which copying is to begin. The character position must be in the range from 1 to *max-int*, inclusive.

char-count

is an integer greater than zero or a variable level that contains an integer greater than zero. *char-count* specifies the number of characters to be copied. The character count must be in the range from 0 to *max-int*, inclusive.

char-addr-2

is an integer greater than zero or a variable level that contains an integer greater than zero. *char-addr-2* specifies the character position at which copying is to end. The character position must be in the range from 1 to *max-int*, inclusive.

Result

#CHARGETV returns nothing.

Considerations

- If you use TO, the character specified by *char-addr-2* is included in the copy: That is, “x TO y” is equivalent to “x FOR (y-x)+1.”
- If you use TO and *char-addr-1* is greater than or equal to *char-addr-2*, or if you use FOR and *char-count* is less than one, no copying occurs.
- If you omit both FOR and TO, one character is copied.
- If *char-addr-1* is less than 1, an error occurs.

- Any part of the specified copy that lies beyond the end of *var-1* is ignored.
- Each logical line contains an end-of-line character that counts as one byte. For variables that contain TACL statements, each metacharacter uses the number of visible characters plus one, including unprintable characters that are subject to change from one TACL RVU to another. Nonmetacharacters use one byte.

Examples

Assume that var is a variable level containing:

```
ABCDEFG
HIJKLMNOPQRST
UVWXYZ
```

1. Either of the invocations:

   ```
   #CHARGETV var var2 2 TO 4 or #CHARGETV var var2 2 FOR 3
   ```

   set var2 to:

   ```
   BCD
   ```

2. The invocation:

   ```
   #CHARGETV var var2 3 FOR 10
   ```

   sets var2 to:

   ```
   CDEFG
   HIJK
   ```

#CHARINS Built-In Function

Use #CHARINS to insert text into a variable level at a specified character address.

```
#CHARINS string char-addr text
```

string

is a string constant or the name of an existing variable level into which text is to be inserted. It must not be in a shared segment or be a DIRECTORY, a STRUCT, or a STRUCT item.

char-addr

is an integer greater than zero or a variable level that contains an integer greater than zero. *char-addr* specifies the character position at which text is to be inserted. The character address must be in the range from 1 to *max-int*, inclusive.

text

is the text to be inserted. The largest valid text length is 32,000 words minus the current contents of the stack. The stack is typically 25,000 words long.

Result

#CHARINS returns nothing.

Considerations

- A text specification can include internal end-of-lines if the entire invocation is enclosed in square brackets, but leading and trailing spaces and end-of-lines are ignored.
- If *char-addr* is beyond the end of the variable level, the text is concatenated with the last line in the variable level.
- Each logical line contains an end-of-line character that counts as one byte. For variables that contain TACL statements, each metacharacter uses the number of visible characters plus one, including unprintable characters that are subject to change from one TACL RVU to another. Nonmetacharacters use one byte.

Examples

Assume that var is a variable level containing:

```
ABCDEFG
HIJKLMNOPQRST
UVWXYZ
```

1. The invocation:

   ```
   #CHARINS var 13 NEW TEXT
   ```

   causes var to contain:

   ```
   ABCDEFG
   HIJKNEW TEXTLMNOPQRST
   UVWXYZ
   ```

2. The invocation:

   ```
   [#CHARINS var 13 NEW TEXT]
   ```

   causes var to contain:

   ```
   ABCDEFG
   HIJKNEW
   TEXTLMNOPQRST
   UVWXYZ
   ```

3. The invocation:

   ```
   #CHARINS var 100 NEW TEXT
   ```

   causes var to contain:

   ```
   ABCDEFG
   HIJKLMNOPQRST
   UVWXYZNEW TEXT
   ```

#CHARINSV Built-In Function

Use #CHARINSV to insert a string into a variable level at a specified character address.

```
#CHARINSV variable-level char-addr string
```

variable-level

is a string constant or the name of an existing variable level into which a string will be inserted. It must not be in a shared segment or be a DIRECTORY, a STRUCT, or a STRUCT item.

char-addr

is an integer greater than zero or a variable level that contains an integer greater than zero. *char-addr* specifies the character position at which the string is to be inserted. The character address must be in the range from 1 to *max-int*, inclusive.

string

is the string constant or the name of a variable level that contains text. string specifies the string to be inserted. It must not be in a shared segment or be a DIRECTORY, a STRUCT, or a STRUCT item. The largest valid string length is 32,000 words minus the current contents of the stack. The amount of remaining space is typically 25,000 words long.

Result

#CHARINSV returns nothing.

Considerations

- The trailing end-of-line in string is suppressed. Leading and trailing spaces are preserved, as are all other end-of-lines.
- If *char-addr* is beyond the end of the variable level, the string is concatenated with the last line in the variable level.
- Each logical line contains an end-of-line character that counts as one byte. For variables that contain TACL statements, each metacharacter uses the number of visible characters plus one, including unprintable characters that are subject to change from one TACL RVU to another. Nonmetacharacters use one byte.

Examples

Assume that var is a variable level containing:

```
ABCDEFG
HIJKLMNOPQRST
UVWXYZ
```

and that var2 is a variable level containing:

```
NEW
TEXT
```

1. The invocation:

   ```
   #CHARINSV var 13 var2
   ```

   causes var to contain:

   ```
   ABCDEFG
   HIJKNEW
   TEXTLMNOPQRST
   UVWXYZ
   ```

2. The invocation:

   ```
   #CHARINSV var 13 "NEW TEXT"
   ```

   causes var to contain:

   ```
   ABCDEFG
   HIJKNEW TEXTLMNOPQRST
   UVWXYZ
   ```

3. The invocation:

   ```
   #CHARINSV var 100 "NEW TEXT"
   ```

   causes var to contain:

```
ABCDEFG
HIJKLMNOPQRST
UVWXYZNEW TEXT
```

#COLDLOADTACL Built-In Function

#COLDLOADTACL indicates whether a TACL process is the cold-load TACL process.

```
#COLDLOADTACL
```

Result

#COLDLOADTACL returns -1 if the TACL process from which it was invoked is the first TACL process started during the cold-load process, and the TACL process has not logged off since it was started. Otherwise, #COLDLOADTACL returns zero.

Consideration

#COLDLOADTACL is used primarily by TACLBASE, which performs certain operations when it is invoked by the cold-load TACL process.

#COMPAREV Built-In Function

Use #COMPAREV to compare one string with another.

```
#COMPAREV string-1 string-2
```

string-1 and *string-2*

are the names of variable levels of a type other than DIRECTORY, text enclosed in quotation marks, or concatenations of such entities. The concatenation operator is '+' (the apostrophes are required).

Result

#COMPAREV returns a nonzero value if the contents of the two arguments are the same; it returns zero if they are different.

Considerations

- The comparison is not case-sensitive; that is, an uppercase character is equal to its lowercase counterpart.
- You can compare any combination of STRUCTs and STRUCT items with each other; such comparisons are case-sensitive.
- You can compare any combination of variable levels that are not STRUCTs or STRUCT items (except type DIRECTORY); such comparisons are not case-sensitive.
- To compare a string to a template, use the #MATCH built-in function.

Example

1. This example shows how a variable level can be compared with quoted text.

   ```
   #PUSH termname term_is_mine
   #SET termname [#MYTERM]
   ```

2. This example returns a nonzero value if #MYTERM is "$MINE".

   ```
   #SET term_is_mine [#COMPAREV termname "$MINE"]
   ```

#COMPUTE Built-In Function

Use #COMPUTE to obtain the value of an arithmetic expression.

```
#COMPUTE expression
```

expression

is an expression using integer values and variable levels, arithmetic operators, and logical operators, as defined in Section 3, Expressions.

Result

- #COMPUTE returns the result of expression; that value can be assigned to a variable level.
- Logical and comparison operators return -1 if the test is true, 0 if the test is false.

Consideration

#COMPUTE performs integer arithmetic; consequently, it discards any fractional part resulting from division. For example, #COMPUTE 2 / 3 yields a zero result.

Example

1. This routine computes the sum of two numbers:

```
?SECTION compute^nums ROUTINE
#FRAME
#PUSH a b c
#SETMANY a b, 1 2
#SET c [#COMPUTE a + b]
#OUTPUTV c
```

2. The routine produces this output:

```
12> compute^nums
3
13>
```

#COMPUTEJULIANDAYNO Built-In Function

Use #COMPUTEJULIANDAYNO to convert a calendar date on or after January 1, 0001, to a Julian day number. A Julian day number is an integral number of days elapsed since January 1, 4713 B.C.

```
#COMPUTEJULIANDAYNO year month day
```

year

is a four-digit number, from 1975 through 9999, representing the year.

month

is a number in the range 1 to 12 representing the month.

day

is a number in the range 1 to 31 representing the day of the month.

Result

#COMPUTEJULIANDAYNO returns a space-separated list of four numbers. The first number is the Julian day number. If an error occurs, TACL sets the first number to -1. The remaining three numbers are error flags that indicate range errors in the three arguments. A flag is 0 if its matching argument (year, month, or day) is within the range for the date element it specifies, or -1 if it is outside the range.

Considerations

- Specifying 14 for the month or 87 for the day, for example, causes #COMPUTEJULIANDAYNO to return a range error in the flag representing that argument. Specifying 31 for a 30-day month or February 29 in a year other than a leap year, for example, causes range errors for both month and day because #COMPUTEJULIANDAYNO is unable to determine which field is actually in error.
- For examples showing the use of time functions, see the *TACL Programming Guide*.

Example

This example shows #COMPUTEJULIANDAYNO output:

```
29> #OUTPUT [#COMPUTEJULIANDAYNO 1994 3 31]
2449078 0 0 0
```

This example shows #COMPUTEJULIANDAYNO error output:

```
30> #OUTPUT [#COMPUTEJULIANDAYNO 1993 4 65]
-1 0 0 -1
```

#COMPUTETIMESTAMP Built-In Function

Use #COMPUTETIMESTAMP to convert a calendar date to a four-word timestamp.

```
#COMPUTETIMESTAMP year month day hour min sec milli micro
```

year

is a four-digit number, from 1975 through 9999, representing the year.

month

is a number in the range 1 to 12 indicating the month.

day

is a number in the range 1 to 31 indicating the day.

hour

is a number in the range 0 to 23 indicating the hour.

min

is a number in the range 0 to 59 indicating the minute.

sec

is a number in the range 0 to 59 indicating the second.

milli

is a number in the range 0 to 999 indicating the millisecond.

micro

is a number in the range 0 to 999 indicating the microsecond.

Result

#COMPUTETIMESTAMP returns a space-separated list of nine numbers. The first number is the four-word timestamp. If an error occurs, TACL sets the first number to -1. The other eight numbers are error flags that indicate range errors in the arguments. An error flag is 0 if its matching argument is within the range for the date/time element it specifies, or -1 if it is outside the range.

Example

This example shows #COMPUTETIMESTAMP output:

```
29> #OUTPUT [#COMPUTETIMESTAMP 1992 3 31 15 37 50 273 146]
211568816270273146 0 0 0 0 0 0 0 0
```

#COMPUTETRANSID Built-In Function

Use #COMPUTETRANSID to convert the separate numeric values for the components of a transaction ID into a single numeric transaction ID.

```
#COMPUTETRANSID system cpu sequence crash-count
```

system

is the system number of the transaction ID.

cpu

is the CPU number of the transaction ID.

sequence

is the sequence number of the transaction ID.

crash-count

is the crash count of the transaction ID.

Result

#COMPUTETRANSID returns a numeric status code indicating the outcome of the conversion:

Code	Condition
-4	Invalid system
-3	Invalid crash count
-2	Invalid CPU
-1	Invalid sequence
0	Successful conversion

Any other value indicates a TACL problem; contact your service provider.

If the status code is zero, the numeric transaction ID follows the status code, separated by a space.

Consideration

Use the #INTERPRETTRANSID built-in function to convert a numeric transaction ID to separate numeric values for the components of the transaction ID.

#CONTIME Built-In Function

Use #CONTIME to break down a three-word timestamp into a seven-number set of date and time components.

```
#CONTIME timestamp
```

timestamp

is a decimal number obtained from #TIMESTAMP or from #FILEINFO / MODIFICATION /.

Result

#CONTIME returns seven numbers representing various components of date and time (year, month, day, hour, minute, second, hundredths of a second), as shown in the following examples.

Examples

1. Assuming #INFORMAT is set to TACL, this example illustrates the use of #CONTIME with #TIMESTAMP:

   ```
   14> #OUTPUT [#CONTIME [#TIMESTAMP]] 1992 06 27 9 28 16 16
   ```

2. This code shows how portions of a timestamp can be extracted and used to create a file name that includes the month and day the file was created:

   ```
   #PUSH date month list
   #SETMANY _ month date, [#CONTIME [#TIMESTAMP]]
   #CHARINSV date 1 month == Insert mm before dd in date
   VARTOFILE list TEMP[date] == Output to a file named TEMPmmdd
   ```

   If, for example, the date were June 10, this code would create a file named TEMP0610.

#CONVERTPHANDLE Built-In Function

Use #CONVERTPHANDLE to convert a process file identifier to a process handle or a process handle to a process file identifier.

```
#CONVERTPHANDLE
  { / PROCESSID / integer-string } |
  { / INTEGERS / process-identifier }
```

integer-string

is a process handle represented by ten integers, each separated by a period.

process-identifier

is a process name or CPU,PIN.

Results

If you specify PROCESSID, #CONVERTPHANDLE returns a process name for the specified integer sequence. If the process is not named, TACL returns the CPU and PIN. The node name is always included. If the specified process handle is not valid, TACL returns an error.

If you specify INTEGERS, #CONVERTPHANDLE returns a process handle, represented as ten integers. If the process does not exist, a null process handle, consisting of the value 65535 in each of the ten integers, is returned.

Considerations

Process handles are the D-series successor to process identifiers (CRTPIDs) for process-control purposes. Process handles can occur in SPI buffers and in operating system messages. TACL provides a process handle in the TACL structure :_COMPLETION^PROCDEATH for completion handling on D-series systems.

Note. The format for a process handle is defined by as part of the TACL software product and is subject to change in future RVUs.

Use the external format of a process handle (ten integers separated by periods) if you need to present a process handle to TACL in external form, as in calls to #SSPUT and #SET.

Examples

1. This function call converts a ten-integer process handle to its corresponding process identifier:

   ```
   10> #CONVERTPHANDLE/PROCESSID/ 512.31.2.18.0.0.11.57728.0.243
   \SYS2.$S
   ```

2. This function call converts a process name to its corresponding process handle:

```
11> #CONVERTPHANDLE / INTEGERS / $S

512.31.2.18.0.0.11.57728.0.243
```

3. This example uses #CONVERTPHANDLE to convert a process handle field in STRUCT s to a process identifier:

```
[#CONVERTPHANDLE / PROCESSID / [s:proc]]
```

where

`proc`	is a process-handle field in the STRUCT named s.
`[s:proc]`	returns the ten-integer format of the process handle, as obtained from an SPI buffer using the #SSGET built-in function.
`[#CONVERTPHANDLE /PROCESSID/ [s:p]]`	returns the process descriptor; for example: \SYSTWO.$PROC.

#CONVERTPROCESSTIME Built-In Function

Use #CONVERTPROCESSTIME to convert the time value obtained by the PROCESSTIME option of #PROCESSINFO.

```
#CONVERTPROCESSTIME process-time
```

process-time

is a numeric expression representing a CPU time.

Result

#CONVERTPROCESSTIME returns a space-separated list of the equivalent number of hours, minutes, seconds, milliseconds, and microseconds in the process-time argument.

Example

Assuming #INFORMAT is set to TACL, this example converts the value (in microseconds) obtained by the PROCESSTIME option of #PROCESSINFO:

```
19> #OUTPUT [#CONVERTPROCESSTIME [#PROCESSINFO
/PROCESSTIME/]]
0 0 4 122 58
```

#CONVERTTIMESTAMP Built-In Function

Use #CONVERTTIMESTAMP to convert a Greenwich mean time (GMT) timestamp to or from a local time-based timestamp on any accessible node in a network. The resulting timestamp is still in four-word format (as returned by #JULIANTIMESTAMP), but its value is adjusted for the time differential between local time and GMT. Local time can be either standard time or civil time (standard time adjusted for daylight saving).

```
#CONVERTTIMESTAMP gmt-timestamp direction [ \node-name ]
```

gmt-timestamp

is the timestamp to be converted.

direction

is a number that represents the direction of conversion:

Number	Conversion
0	GMT to local civil time (LCT)
1	GMT to local standard time (LST)
2	LCT to GMT
3	LST to GMT

\node-name

is the name of the system where conversion is to be done.

Result

#CONVERTTIMESTAMP returns a numeric error code. If the conversion is successful, the error code (zero) is followed by a space and the converted timestamp.

The codes are:

Code	Condition
-2	Impossible LCT
-1	Ambiguous LCT
0	No errors; successful conversion
1	Daylight-saving time (DST) range error
2	DST table not loaded
>2	File-system error (attempting to reach \ node-name)

Example

This example obtains the current Greenwich mean time, converts the timestamp to local civil time (LCT), and displays the GMT and LCT in component form:

```
?SECTION convtime ROUTINE
#FRAME
#PUSH gmttime, loctime, err
#SET gmttime [#JULIANTIMESTAMP]
#SETMANY err loctime , [#CONVERTTIMESTAMP gmttime 0]
SINK [IF [err] |THEN|
    #OUTPUT Error occurred: [err]
  |ELSE|
    #OUTPUT GMT = [#INTERPRETTIMESTAMP gmttime]
    #OUTPUT LCT = [#INTERPRETTIMESTAMP loctime]
]
#UNFRAME
```

When the preceding routine is invoked, TACL displays:

```
GMT = 2448713 1992 3 31 0 53 47 174 708
LCT = 2448712 1992 3 30 16 53 47 174 708
```

#CREATEFILE Built-In Function

Use #CREATEFILE to create an unstructured file.

```
#CREATEFILE [ / option [ , option ] / ] file-name
```

option can be:

EXTENT *num*

the number of data pages allocated for the primary and all secondary extents.

- For format 1 files, specify *num* as an integer in the range from 1 through 65535.
- For format 2 files, specify *num* as an integer in the range from 1 through 512000000.

The default value is 2.

FILEFORMAT *num*

the file format.

- For format 1 files, specify *num* as 1.
- For format 2 files, specify *num* as 2.

The default value is 1.

file-name

the name of the file to be created.

If you specify only the volume name, a temporary file is created on the specified volume. To retrieve this name, use the FILENAMEV option.

FILENAMEV *variable-level*

the name of an existing variable level into which the filename is to be returned. The previous contents of the variable are lost if the file is created successfully. The variable must not be in a shared segment and must not be a DIRECTORY, a STRUCT, a SUBSTRUCT, or a STRUCT item.

Variable levels are defined in Section 4, Variables.

PHYSICALVOLUME *physical-volume*

the physical volume on which a logical file is to be created. If no physical volume is specified in *file-name*, the default volume name for the TACL process in effect at the time of the request is used.

If a logical volume is specified in the file name for file creation, the physical volume on which the file is created is chosen by the system. The physical-volume parameter overrides this selection.

The PHYSICALVOLUME option should be used only if no physical volume is specified in *file-name*. Otherwise, a file-system error is returned.

Result

#CREATEFILE returns zero if it is successful; otherwise, it returns a file-system error indicating the reason for the failure.

Considerations

- If you specify only the volume for the filename, a temporary file is created on the specified volume. You should also specify the FILENAMEV to retrieve the temporary file name.
- The PHYSICALVOLUME option should be used only if a logical volume was specified for the filename. Otherwise a file-system error is returned.
- If the system does not support the PHYSICALVOLUME option, error 561 is returned.
- Various failures on a system with a DSM/Storage Manager might result in the inaccessibility of a logical file.

Example

```
#CREATEFILE /EXTENT 2,PHYSICALVOLUME $mg/ $v.a.b
```

Specifies that logical file, $v.s.f, be created with a two-page extent size on physical volume $mg.

#CREATEPROCESSNAME Built-In Function

Use #CREATEPROCESSNAME to create a unique process name that is not in use on the local system. This function invokes the CREATEPROCESSNAME operating system procedure.

```
#CREATEPROCESSNAME
```

Result

#CREATEPROCESSNAME returns a unique, unused process name in one of the forms $*Xnnn*, $*Ynnn*, or $*Zxxx*, where *n* is any numeric character and *x* is any alphanumeric character.

#CREATEREMOTENAME Built-In Function

Use #CREATEREMOTENAME to create a unique process name not in use on the specified system. This function invokes the CREATEREMOTENAME operating system procedure.

```
#CREATEREMOTENAME \node-name
```

\node-name

is the name of an available system to be checked for processes in use.

Result

#CREATEREMOTENAME returns a unique, unused process name in one of the forms $*Xnnn*, $*Ynnn*, or $*Zxxx*, where n is any numeric character and x is any alphanumeric character. If *node-name* does not exist, TACL returns an error. If *node-name* is the name of the local system, #CREATEREMOTENAME returns a process name for the local system.

#DEBUGPROCESS Built-In Function

Use #DEBUGPROCESS to invoke the default debugger, DEBUG or INSPECT, for a specified process. #DEBUGPROCESS invokes the DEBUGPROCESS operating system procedure.

```
#DEBUGPROCESS [ / NOW / ]
  [\node-name.]
  {$process-name | cpu,pin}
  [ , TERM [\node-name.]$terminal-name ]
```

NOW

specifies that debugging is to begin immediately. This option is available to super-group users only.

\node-name

is the system where the process resides.

$*process-name*

is the name of the process or process pair.

cpu,pin

is the CPU number and process number of the process.

TERM [*node-name*.]$*terminal-name*

is the terminal where interactive debugging is to take place. It also becomes the home terminal of the specified process. If *\node-name* is the current default system, you can omit it. If you omit this option, DEBUG or INSPECT uses the existing home terminal of the specified process.

Result

#DEBUGPROCESS returns zero if it is successful; otherwise, it returns the error code returned by the call to the DEBUGPROCESS procedure.

Considerations

- If you are not a super-group member or a group manager, you can debug only those processes whose process accessor ID matches your user ID; you must have read access to the program file.
- If you are a group manager, you can debug any process whose process accessor ID matches any user ID in your group; you must have read access to the program file.

- If you are the super ID, you can debug any process. Only the super ID can debug privileged processes.
- The process to be debugged does not enter the debug state until it executes its next instruction in the user code space; the process cannot enter the debug state while executing system code. If you enter the NOW option, however, the process enters the debug state immediately. To use this option, you must have a group ID of 255.
- To set or obtain the current value of the INSPECT flag, which determines the default debugger, use the #INSPECT built-in function.

H-Series Usage

The program DEBUG is not available for use on systems running H-series software.

The DEBUG command invokes a debugger, it can be Inspect, Native Inspect (eInspect, which is not in the family of Inspect debuggers), or Visual Inspect.

The rules about which debugger gets invoked are approximately the same as for the RUND command. That is, if the INSPECT attribute is set ON anywhere (in the object file during compilation, or on the TACL command line using the SET command), you will get a debugger in the Inspect family (either Inspect or VI), unless of course neither of these debuggers is available, and then you get the default debugger, eInspect. If the Inspect attribute is OFF, you get Native Inspect (eInspect).

Inspect is invoked only for TNS accelerated/interpreted programs (never for TNS/E native programs), while Visual Inspect can handle both of these. Native Inspect handles only TNS/E native programs and snapshots.

#DEF Built-In Function

Use #DEF to define a variable.

```
#DEF variable {
  { ALIAS | DELTA | MACRO | ROUTINE | TEXT} enclosure}
  DIRECTORY [ segment-spec ]
  STRUCT structure-body
}
```

variable

is the name of the variable. If the variable exists, TACL pushes it, creating a new top level. If the variable does not exist, TACL creates it.

ALIAS

specifies that *variable* is a command alias.

DELTA

specifies that *variable* is to contain #DELTA commands.

MACRO

specifies that *variable* is a TACL macro.

ROUTINE

specifies that *variable* is a TACL routine.

TEXT

specifies that *variable* is to contain plain text.

enclosure

has this form (any other labels and associated text are ignored):

|BODY| [*text*]

It defines the contents of the variable.

DIRECTORY [*segment-spec*]

specifies that *variable* is a directory. Use DIRECTORY to create a directory that can contain a hierarchy of variables or to attach a segment. If you specify *segment-spec*, TACL attaches the segment. If you omit *segment-spec*, TACL pushes the named variable (in the segment in which it is defined) but does not associate it with a segment file.

segment-spec

has the form { PRIVATE | SHARED } *file-name*

PRIVATE

specifies that the creator of the segment file has read and write access to the file and that no other process may open it.

SHARED

specifies that the segment file is a read-only file and that other processes may open it for read access.

file-name

is the name of a TACL segment file. If *file-name* does not exist, TACL creates it and initializes it as an empty TACL segment.

STRUCT *structure-body*

specifies that *variable* represents a structure.

structure-body

is a set of declarations for data, substructures, FILLER bytes, or redefinitions, as described in Section 4, Variables. All internal square brackets in the body are expanded before the structure is declared.

Result

#DEF returns nothing.

Considerations

- If you specify a segment file in a DIRECTORY definition, the segment file must reside on the local system (where the TACL process is executing).
- If the #DEF declaration contains a |BODY| label, it must be enclosed in square brackets. For a DIRECTORY or STRUCT definition, square brackets are required only if the entire #DEF cannot be contained in one line and you prefer not to use the ampersand (&) continuation character.
- The |BODY| label, used in the enclosure that defines ordinary variables, is not used in a DIRECTORY or STRUCT definition.
- #DEF does not always behave exactly the same as a #PUSH-#SET combination: For example, #SET A BB[CCC]DDDD invokes [CCC] to obtain the contents of the variable level CCC before assigning the result to the variable level A. On the other hand, [#DEF A type |BODY| BB[CCC]DDDD] assigns the value BB[CCC]DDDD to A, including the brackets, leaving the expansion to be done when A is invoked. This may cause unexpected results, especially if type is DELTA.
- Do not try to use #DEF on the root directory (:). If you try this, TACL returns “*ERROR* Cannot push or pop the root segment's root.” In addition, to avoid losing

standard functionality from your TACL environment, do not use #DEF on the directories supplied as part of the TACL software product (such as UTILS).

- You cannot attach more than 50 segment files.

Examples

1. This example defines a variable, RTN, as a ROUTINE that accepts no arguments and displays a “thank you” message:

```
[#DEF rtn ROUTINE
  |BODY|
    SINK [#ARGUMENT END]
    #OUTPUT Thank you!
]
```

2. The next example sets up the alias P for the PERUSE command:

```
[#DEF p ALIAS
  |BODY|
  PERUSE
]
```

3. These examples declare structures:

```
[#DEF inventory STRUCT
  BEGIN
    INT item;
    INT price;
    INT quantity;
  END;
]

[#DEF obsolete^stuff STRUCT
  LIKE inventory;
]
```

4. This examples define directories:

```
PURGE segfl
#DEF :myseg DIRECTORY PRIVATE segfl
#DIR :our^seg DIRECTORY SHARED $system.segs.segfile
#DEF :dir DIRECTORY
```

#DEFAULTS Built-In Variable

Use #DEFAULTS to set or obtain the saved defaults (set by the DEFAULT program), the current defaults (set by the VOLUME and SYSTEM commands), or both.

```
#DEFAULTS [ / option [ , option ] / ]
```

option

is either of these:

CURRENT

specifies your current defaults.

SAVED

specifies your logon defaults.

Result

#DEFAULTS returns the file-name defaults you request. If you specify both options, the defaults appear in a space-separated list in the order in which you presented the requests. If you omit both options, #DEFAULTS returns the current defaults.

Considerations

- When you first log on, #DEFAULTS is initialized to your saved default volume and subvolume (current default and saved default are the same at this time).
- Use #PUSH #DEFAULTS (or PUSH #DEFAULTS) to save a copy of your current file-name defaults.
- Use #POP #DEFAULTS (or POP #DEFAULTS) to replace your current file-name defaults with the previously pushed defaults.
- Use #SET #DEFAULTS (or SET VARIABLE #DEFAULTS) to assign your current defaults to a specified subvolume.

The syntax for #SET #DEFAULTS is:

```
#SET #DEFAULTS subvolume-name
```

If you change your defaults to include a node name, you can get them back to local form only by using the #SYSTEM built-in function or the SYSTEM command.

Example

Assuming #INFORMAT is set to TACL, this example displays both your current and saved defaults:

```
12> #OUTPUT [ #DEFAULTS /SAVED, CURRENT/ ]
$BUNK.HOUSE $OPEN.RANGE
```

#DEFINEADD Built-In Function

Use #DEFINEADD to add a DEFINE to the TACL context, or to replace an existing DEFINE, using the attributes in the working set. #DEFINEADD invokes the DEFINEADD operating system procedure.

```
#DEFINEADD define-name [ flag ]
```

define-name

is the name of the DEFINE to be added.

flag

is a flag that specifies what to do if a DEFINE with the same name already exists. It can have either of these values:

0 Add a DEFINE if it does not exist. Do not replace the existing DEFINE.

1 Replace an existing DEFINE.

If you omit flag, 0 is assumed.

Result

#DEFINEADD returns a numeric error code indicating the outcome of the DEFINEADD procedure; zero indicates success. See Appendix B, Error Messages for a list of DEFINE-oriented error codes.

Consideration

When a backup TACL process takes over, TACL deletes existing assignments.

#DEFINEDELETE Built-In Function

Use #DEFINEDELETE to delete a DEFINE from the TACL context. This function invokes the DEFINEDELETE operating system procedure.

```
#DEFINEDELETE define-name
```

define-name

is the name of the DEFINE to be deleted.

Result

#DEFINEDELETE returns a numeric error code indicating the outcome of the DEFINEDELETE procedure. Zero indicates success. See Appendix B, Error Messages for a list of DEFINE-oriented error codes. If an error occurs, the DEFINE is not deleted.

#DEFINEDELETEALL Built-In Function

Use #DEFINEDELETEALL to delete all DEFINEs except =_DEFAULTS from the TACL context (=_DEFAULTS cannot be deleted). This function invokes the DEFINEDELETEALL operating system procedure.

```
#DEFINEDELETEALL
```

Result

#DEFINEDELETEALL returns a numeric error code indicating the outcome of the DEFINEDELETEALL procedure; zero indicates success. See Appendix B, Error Messages for DEFINE-oriented error codes.

#DEFINEINFO Built-In Function

Use #DEFINEINFO to set or obtain information about a specified DEFINE. The function invokes the DEFINEINFO operating system procedure.

```
#DEFINEINFO define-name
```

define-name

is the name of the DEFINE for which information is desired.

Result

#DEFINEINFO returns a numeric error code indicating the outcome of the DEFINEINFO procedure; zero indicates success. See Appendix B, Error Messages for a list of DEFINE-oriented error codes.

If the error code is 0, the following are also included as a space-separated list:

- The class of the DEFINE
- The name of the primary attribute (dependent on the class of the DEFINE)
- The (possibly empty) value of the selected attribute

#DEFINEMODE Built-In Variable

Use #DEFINEMODE to obtain the current setting of DEFMODE (described with the SET DEFMODE command), which controls the use and propagation of DEFINEs.

```
#DEFINEMODE
```

Result

#DEFINEMODE returns the current DEFMODE setting: OFF or ON.

Considerations

- When you first log on, #DEFINEMODE is initialized to ON.
- #DEFINEMODE is set to ON when TACL is started and whenever a LOGON is done from the logged-off state.
- Use #PUSH #DEFINEMODE (or PUSH #DEFINEMODE) to save the current DEFMODE status.
- Use #POP #DEFINEMODE (or POP #DEFINEMODE) to restore #DEFINEMODE to its previous setting.
- Use #SET #DEFINEMODE (or SET VARIABLE #DEFINEMODE) to establish whether DEFINEs can be used by TACL and the processes it starts.

  The syntax for #SET #DEFINEMODE is:

```
#SET #DEFINEMODE { OFF | ON }
```

`OFF`

disables the use of DEFINEs in TACL and the propagation of DEFINEs to new processes started by TACL (such processes have an initial DEFMODE setting of OFF).

`ON`

enables the use of DEFINEs in TACL and causes it to propagate all DEFINEs in use to any process it starts (such processes have an initial DEFMODE setting of ON).

#DEFINENAMES Built-In Function

Use #DEFINENAMES to find the names of DEFINEs that match a specified DEFINE template.

```
#DEFINENAMES define-template
```

define-template

is a template indicating the DEFINE names to search for. The template can include these template characters:

```
* matches zero or more characters

? matches any single character
```

The template can consist entirely of two template characters:

```
=* or ** matches all DEFINE names
```

Result

#DEFINENAMES returns a (possibly empty) space-separated list of all DEFINE names that match the DEFINE template.

#DEFINENEXTNAME Built-In Function

Use #DEFINENEXTNAME to obtain the name of the DEFINE that follows the specified DEFINE in the sequence established by the operating system. The function invokes the DEFINENEXTNAME operating system procedure.

```
#DEFINENEXTNAME [ define-name ]
```

define-name

specifies the name from which to begin searching for the next DEFINE name. If you omit it, the function returns the first DEFINE name.

Result

#DEFINENEXTNAME returns a numeric error code indicating the outcome of the DEFINENEXTNAME procedure; zero indicates success. See Appendix B, Error Messages for a list of DEFINE-oriented error codes.

If the error code is 0, it is followed by a space and the name of the next DEFINE in sequence.

#DEFINEREADATTR Built-In Function

Use #DEFINEREADATTR to obtain the current value of a specified attribute in the TACL context or in the working set. This function invokes the DEFINEREADATTR operating system procedure.

```
#DEFINEREADATTR
  { define-name | _ }
  { attribute-name | cursor-mode }
```

define-name

is the name of the DEFINE in the TACL context from which an attribute is to be read.

_

An underscore in place of *define-name* specifies that the attribute is to be read from the current working set.

attribute-name

is the name of the attribute to be read.

cursor

is a numeric pointer to the attribute to be read; zero indicates the first attribute in the DEFINE. You can use cursor, along with mode, in place of *attribute-name* as an alternative method of attribute specification.

mode

is a numeric indicator used with cursor; it can be any of these:

0 Search only attributes that are present

1 Search present attributes, plus required attributes not present

2 Search present attributes, plus required and optional attributes not present

Result

#DEFINEREADATTR returns a numeric error code indicating the outcome of the DEFINEREADATTR procedure; zero indicates success. See Appendix B, Error Messages for a list of DEFINE-oriented error codes.

If the error code is 0 or 2061 (no more attributes), the following are also included as a space-separated list:

- The cursor value for the next attribute in sequence, consistent with the mode specification. This is zero if an attribute name, rather than a cursor, was supplied as an argument.

- A number indicating the type of the attribute: 0 if it is optional, 1 if it is defaulted, or 2 if it is required.
- A number indicating the condition of the attribute: 1 if it was inconsistent the last time attributes were validated, 0 if it was consistent.
- The name of the attribute whose value is being returned.
- The (possibly empty) value of the selected attribute.

Consideration

An error code of 2061 can be considered as indicating a successful operation, but if you are using a loop to read successive attributes, receipt of the “no more attributes” code should be a signal to terminate the loop.

#DEFINERESTORE Built-In Function

Use #DEFINERESTORE to create or replace an active DEFINE, or replace the working set, with the contents of a DEFINE previously saved with #DEFINESAVE.

```
#DEFINERESTORE [ / option [ , option ] / ] buffer
```

option

is either of these:

REPLACE

specifies that if a DEFINE with the same name as the saved DEFINE exists, the saved DEFINE is to replace the existing DEFINE. If no DEFINE with the same name exists, an error occurs.

If you omit this option, and no other DEFINE exists with the same name, the saved DEFINE is added; otherwise, an error occurs.

WORK

specifies that the saved DEFINE is to replace the working set; if the REPLACE option is also present, it is ignored. If you omit this option, the presence or absence of the REPLACE option governs restoration of the DEFINE.

buffer

is the name of a STRUCT containing a previously saved DEFINE. The definition of the STRUCT, except for its length, is irrelevant.

Result

#DEFINERESTORE returns a space-separated list consisting of a numeric error code, the name of the saved DEFINE, and a numeric consistency-check result. Error codes are listed in Appendix B; zero indicates a successful operation. Consistency-check numbers are shown in Table 8-7 on page 8-185; zero indicates no consistency errors.

Considerations

- The buffer must contain a valid internal form of a DEFINE, as created by #DEFINESAVE. If the buffer appears not to contain a valid DEFINE, #DEFINERESTORE returns an error and does not restore the buffer contents.
- If the buffer is to be restored as an active DEFINE and no error occurs, the DEFINE is restored; if the error code is not zero, the DEFINE is not restored. In either case, the working set and the background set remain unchanged.
- If the buffer is to be restored to the working attribute set and the error code is anything but 0, 2057, 2058, or 2059, the working set remains unchanged.

However, the saved DEFINE is restored to the working set even if it is incomplete, inconsistent, or invalid.

- If you save the =_DEFAULTS DEFINE, you must use the REPLACE option when restoring it. Because the =_DEFAULTS DEFINE always exists, it cannot be added.

#DEFINERESTOREWORK Built-In Function

Use #DEFINERESTOREWORK to restore the DEFINE working set from the background set. This function invokes the DEFINERESTOREWORK operating system procedure.

```
#DEFINERESTOREWORK
```

Result

#DEFINERESTOREWORK returns a numeric error code indicating the outcome of the DEFINERESTOREWORK procedure; zero indicates success. See Appendix B, Error Messages for DEFINE-oriented error codes.

#DEFINESAVE Built-In Function

Use #DEFINESAVE to save a copy of an active DEFINE, or the working set, for later restoration by #DEFINERESTORE.

```
#DEFINESAVE [ / WORK / ] define-name buffer
```

WORK

specifies that the working set is to be saved with the specified DEFINE name. If you omit this option, the DEFINE with the specified name is saved.

define-name

is the name of the DEFINE to be saved.

buffer

is the name of a STRUCT that is to receive a copy of the DEFINE. The definition of the STRUCT, except for its length, is irrelevant.

Result

#DEFINESAVE returns a numeric error code, a space, and the number of bytes required to save the DEFINE. The error codes are:

Code	Condition
0	No error
2049	Invalid DEFINE name
2051	DEFINE does not exist
2052	Unable to obtain file system buffer space
2053	Unable to obtain physical memory
2054	Bounds error on buffer, DEFINE, or saved length
2066	Parameter missing
2076	Buffer too small
2077	Buffer or DEFINE is in invalid segment

If the error code is any of the above except zero, the DEFINE is not saved.

These codes are warnings only, and are used only when the working set is being saved; it is saved even if one of the following appears.

Code	Condition
2057	Working set is incomplete.
2058	Working set is inconsistent.
2059	Working set is invalid.

Considerations

- The internal form of the DEFINE is placed in the buffer if it is large enough to hold it.
- If the buffer is too small, an error occurs; you can use the length field of the result to determine how large the buffer should have been. The following are estimates of the maximum buffer size needed for each DEFINE class:

Class	Estimated Size
CATALOG	202 bytes
DEFAULTS	2300 bytes
MAP	118 bytes
SORT	740 bytes
SPOOL	274 bytes
SUBSORT	226 bytes
TAPE	392 bytes

- If the buffer is larger than the DEFINE, the unused portion of the buffer contains unpredictable data.
- You should not modify the data in the buffer in any way; if it is modified, #DEFINERESTORE might not be able to restore it.
- If the working set is to be saved, the DEFINE name can be that of an active DEFINE; the working set is saved regardless.
- The working set can be saved even if it is inconsistent, invalid, or incomplete; an appropriate error is indicated in the result, however.

#DEFINESAVEWORK Built-In Function

Use #DEFINESAVEWORK to save the DEFINE working set in the background set. This function invokes the DEFINESAVEWORK operating system procedure.

```
#DEFINESAVEWORK
```

Result

#DEFINESAVEWORK returns a numeric error code indicating the outcome of the DEFINESAVEWORK procedure; zero indicates success. See Appendix B, Error Messages for a list of DEFINE-oriented error codes.

#DEFINESETATTR Built-In Function

Use #DEFINESETATTR to modify the value of an attribute in the working set. #DEFINESETATTR 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. This function invokes the DEFINESETATTR operating system procedure.

```
#DEFINESETATTR attribute-name [ attribute-value ]
```

attribute-name

is the name of the attribute to be set.

attribute-value

is the value to be assigned to the attribute. If you omit it, DEFINESETATTR resets the attribute to its default value; if there is no default value, the procedure deletes the attribute from the working set.

Result

#DEFINESETATTR returns a numeric error code indicating the outcome of the DEFINESETATTR procedure; zero indicates success. See Appendix B, Error Messages for a list of DEFINE-oriented error codes.

Considerations

- If the value to be assigned is an unqualified file name, the DEFINESETATTR procedure supplies the file-name qualification from the default volume information.
- If the CLASS attribute is set, the working set is reinitialized with the attributes of the new class and their default values (even if the CLASS attribute value is unchanged).
- Required attributes cannot be reset.

#DEFINESETLIKE Built-In Function

Use #DEFINESETLIKE to initialize the working set with the attributes of an existing DEFINE. This function invokes the DEFINESETLIKE operating system procedure.

```
#DEFINESETLIKE define-name
```

define-name

is the name of the DEFINE whose attributes are to be copied to the working set.

Result

#DEFINESETLIKE returns a numeric error code indicating the outcome of the DEFINESETLIKE procedure; zero indicates success. See Appendix B, Error Messages for a list of DEFINE-oriented error codes.

Consideration

#DEFINESETLIKE deletes existing attribute values in the working set. You can save attributes in the background set by using #DEFINESAVEWORK before invoking this function.

#DEFINEVALIDATEWORK Built-In Function

Use #DEFINEVALIDATEWORK to check the DEFINE working set for consistency. This function invokes the DEFINEVALIDATEWORK operating system procedure.

```
#DEFINEVALIDATEWORK
```

Result

#DEFINEVALIDATEWORK returns a numeric error code indicating the outcome of the DEFINEVALIDATEWORK procedure, followed by a space and a check number. If the error code is 0, the working set is consistent. See Appendix B, Error Messages for a list of DEFINE-oriented error codes.

The check number is 0 unless the error code is 2058 (working set is inconsistent), in which case the code value depends on the DEFINE class. See Table 8-7 on page 8-185 for a list of check numbers.

#DELAY Built-In Function

Use #DELAY to cause TACL to delay for a specified number of centiseconds.

```
#DELAY centiseconds
```

centiseconds

is the number of centiseconds for TACL to delay.

Result

#DELAY returns nothing.

Consideration

- You cannot end the #DELAY built-in function using the BREAK key if you specify #BREAKMODE DISABLE or #BREAKMODE POSTPONE. You can end the #DELAY built-in function using the BREAK key only when #BREAKMODE ENABLE is specified.
- If I/O is pending or outstanding on a TACL IN file, you cannot break the delay.

Examples

This list illustrates #DELAY values:

```
#DELAY 100      means delay 1 second
#DELAY 6000     means delay 1 minute
#DELAY 30000    means delay 5 minutes
```

#DELTA Built-In Function

#DELTA, the TACL character editor, allows you to perform complex operations on text. Although it is mainly for editing text from within macros and routines, you can use #DELTA interactively as well. Using #DELTA interactively allows you to observe how #DELTA works and lets you debug existing #DELTA macros.

You can also use #DELTA to read text from a file, modify the text and write the modified text to another file.

The #CHARxxx and #LINExxx string-handling functions perform text operations that are similar to #DELTA capabilities. You might find that you can avoid using #DELTA altogether, doing all the character processing you need-much more simply-through the use of the #CHARxxx and #LINExxx string-handling functions.

```
#DELTA [ / COMMANDS variable-level / ] [ text ]
```

variable-level

is a variable level whose type must be DELTA. #DELTA executes commands from this variable level.

text

is a text string that is inserted into the #DELTA buffer before #DELTA starts.

Result

#DELTA returns whatever is left in the #DELTA buffer after editing.

Considerations

- If you include the COMMANDS option, #DELTA executes the commands in *variable-level* and exits.
- If you omit the COMMANDS option, #DELTA accepts commands interactively from the terminal until it receives two consecutive CTRL-y characters.

A detailed description of #DELTA operations and commands follows.

Most of the examples that will be given are interactive (the COMMANDS option, which allows you to name a variable level containing a set of #DELTA commands, is not used in this discussion). Using #DELTA interactively allows you to observe how #DELTA operates, to work on individual parts of #DELTA command variables before formally defining them as a whole, and to debug existing #DELTA macros.

The #DELTA prompt appears in the form #DELTA n> ; the prompt is displayed only when you use #DELTA interactively. At the #DELTA prompt, you can enter #DELTA commands.

About #DELTA Commands

Most #DELTA commands are one or two characters long; #DELTA allows you to enter as many commands as you want in a command string. The RETURN character can be a part of a command or a separator between commands. Commands can be separated by many spaces or no spaces. Because most of the #DELTA commands are only one character long, spaces are important to writing legible command strings.

Many commands require one or two values to indicate the range of the text that is affected by the command. #DELTA uses a postfix notation; that is, you enter the operands (the information on which #DELTA operates) before entering the command. Some commands return values; you can use the returned values from one command as input values for the next command.

The RETURN character can be part of a #DELTA command string, so it is not used as a command-line terminator; instead, you use CTRL-y to end a command line. Whenever you type CTRL-y, #DELTA responds with EOF!.

These rules apply only to the interactive use of CTRL-y:

- Use CTRL-y to end command strings.
- You must use CTRL-y at a #DELTA n> prompt. If you use CTRL-y anywhere else in a command line, the commands on the line containing the CTRL-y are lost.
- If you use CTRL-y when there is no command string, #DELTA exits and returns the contents of the buffer as the result of the #DELTA function.

In the examples that will be given, CTRL-y is not shown (it does not appear on your terminal). However, because the system responds to a CTRL-y with EOF!, wherever you see EOF! in an example you can assume that CTRL-y was entered at that point. For example:

```
6> #DELTA This is a test string
#DELTA 7> ht
#DELTA 7> EOF!
This is a test string
#DELTA 8> ht EOF!

#DELTA This is a test string expanded to:

This is a test string
8> !6
8> DELTA This is a test string
#DELTA 9> v
#DELTA 9> EOF!
This is a test string(.)
#DELTA 10> EOF!

#DELTA This is a test string expanded to:

This is a test string
```

Comments

Comments help to produce readable #DELTA code. A #DELTA comment is an exclamation point (!) followed, optionally, by explanatory text to the end of the line. You can also use the standard TACL “{ }” and “==” comments within #DELTA. (Using TACL comments could be safer because #DELTA ignores all text within comments; you can also continue them over additional lines by using ampersands. However, #DELTA parses “!” comments, so errors within those comments are possible.)

The Buffer

The main work area in #DELTA is a buffer that can contain up to 30,000 characters. However, there could be fewer than 30,000 characters available because of ongoing TACL activity. (At the time #DELTA exits, not more than 15,000 characters can remain in the buffer or a “Text buffer overflow” error occurs.) Use the HT command to display the contents of the buffer; for example:

```
10> [#DELTA These characters are the
10> contents of the buffer]
#DELTA 11> HT
#DELTA 11> EOF!
These characters are the
contents of the buffer
```

#DELTA provides ways of moving all or part of the buffer to or from files or variable levels. All editing is done on the text in the buffer.

#DELTA does no character translation during file I/O or while accessing variable levels; all data access is done in the PLAIN mode. For example, when #DELTA reads a file, square brackets and vertical bars remain ordinary text; they do not become TACL invocation characters. Comments are not eliminated, nor do ampersands function as continuation characters.

When #DELTA reads a variable level containing a routine or a macro, special internal character sequences appear in the buffer. In particular, a square bracket or vertical bar appears as a pair of bytes, the second of which is the square bracket or vertical bar (the first is RVU-dependent and subject to change); remember this point when counting characters.

The Pointer

The pointer indicates your current position in the buffer. Some commands use the pointer value; others modify the position of the pointer. The #DELTA pointer points between characters. When you use a command that depends on the pointer value, #DELTA operates on the characters relative to the pointer position.

You can display the current pointer value by entering a period followed by an equal sign (.=). The V (view) command shows lines in the buffer and indicates by (.) the position of the pointer within the buffer; for example:

```
14> #DELTA Test string
#DELTA 15> .=
```

```
#DELTA 15> EOF!
11
#DELTA 16> v
#DELTA 16> EOF!
Test string(.)
```

When #DELTA is first invoked, the pointer points to the end of the buffer.

The X and Y Registers

As mentioned, you enter the operand values for a #DELTA command before entering the command. When you enter the values, #DELTA stores them in two internal registers named X and Y. When you first enter a value, it is stored in the X register. The comma command (,) moves the X register value into the Y register and clears the X register; you can then enter a new X register value. (If you enter another comma and another new X register value, the first Y register value is lost.)

If a command returns one value, the value is stored in the X register. If a command returns two values, they are stored in the X and Y registers; the individual command descriptions indicate the register in which each value is stored. Commands that do not return values clear the X and Y registers.

#DELTA has several shorthand commands for loading the X and Y registers. The B command loads a 0 into the X register; 0 is the beginning of the buffer. The Z command loads the number of characters in the buffer into the X register; thus, Z also specifies the last position in the buffer. To type out the entire buffer, you can enter:

```
18> #DELTA This is a test string
#DELTA 19> b,zt
#DELTA 19> EOF!
This is a test string
#DELTA 20> EOF
#DELTA This is a test string expanded to:

This is a test string
```

“B,” in the preceding example loads zero into the Y register; “Z” loads the size of the buffer into the X register. The H command does both of those functions. So B,Z T and H T are equivalent.

The period command (.) loads the current pointer position into the X register. The dollar sign command ($) clears the X and Y registers, as does a RETURN between commands.

Specifying Ranges of Text

Many of the #DELTA commands operate on ranges of text. There are two ways you can specify a range of text to these commands:

- If you specify a range of text using an X register value with an empty Y register, the range is the characters between the pointer position and x number of end-of-lines. (EOLs) For example, 1T means type the rest of the current line (one EOL); 2T means type the rest of the current line and the next line (two EOLs). A zero means

the range between the current pointer position and the beginning of the current line.

- If you specify both X and Y register values, the range is the characters between the absolute locations y and x. For example, 4,8T means type the characters between the fourth character and eighth character in the buffer (this can include end-of-line characters).

X Register Arithmetic

You can do mathematical operations on the contents of the X register. The available operators are:

+ n	Addition
- n	Subtraction
* n	Multiplication
/ n	Division

In each case, n can be a number or it can be any of these:

.	The current pointer
B	The beginning of the buffer
FL	The length of the text last inserted by an I command, or the length of the string last found by an S command
Z	The length of the buffer

For example, these commands display the text from the current pointer position to the character that is ten characters to the right of the current pointer position (note that the first command simply displays the pointer location within the line):

```
#DELTA 24> v
#DELTA 24> EOF!
This is (.)a test string
#DELTA 25> .,.+10t
#DELTA 25> EOF!
a test str
```

Other examples of X register arithmetic are:

```
[#DEF zerofill DELTA |BODY|
  0J == Jump to beginning of buffer.
  6-Z,48I == Insert enough zeros to make 6 characters long.
] == End DEF

[#DEF truncate DELTA |BODY|
  Z-79 ?G == If text is longer than 79 characters ...
  79,ZK == ... delete excess characters.
  '
] == End DEF
```

#DELTA Commands

This subsection describes the commands that you can use in #DELTA. Table 9-5 is a summary of all #DELTA commands.

Table 9-5. Summary of #DELTA Commands (page 1 of 2)

Command	Description
$	Clears X and Y registers
,	Moves X into Y
.	Gets pointer position
=	Displays X or Y,X
\	Converts value between text and X register
?	Identifies condition to be tested
:?	Negative condition
’	Ends condition
<	Begins iteration
;	Exits from iteration
>	Ends iteration
^\	Exits from current macro defined by M command
A	Converts character to its ASCII code
B	Sets X to zero (Beginning of buffer)
C	Moves pointer X Characters from current position
D	Deletes X characters starting at current pointer position
EI	Opens a file for Input
EO	Opens a file for Output
FC	Sets Case of specified range to lowercase
@FC	Sets Case of specified range to uppercase
FE	Tests a variable level for Emptiness
FF	Gets Frame number associated with a variable level
FG	Compares range (Group) of text to a variable level
FL	Sets X to Length of last text inserted by I command or of last string found by S command
FO	Pops a variable
FT	Gets or sets a variable Type
FU	Pushes a variable
G	Gets text of variable level into buffer
H	Sets Y to zero, X to buffer length (wHole buffer)
I	Inserts text at pointer position

Table 9-5. Summary of #DELTA Commands (page 2 of 2)

Command	Description
J	Jumps pointer to absolute character position
K	Deletes (Kill) lines of text
L	Moves pointer over X Lines
M	Invokes a variable level as a Macro
P	Puts a range of text into the output file
Q	Loads numeric variable level into X register
S	Searches for a string in text
T	Types out text
U	Unloads X register into a variable level
V	Views text, including pointer position
X	EXtracts range of text into a variable level
Y	Reads (Yank) lines from input file
Z	Gets buffer size

#DELTA commands can be grouped into four categories:

- Text manipulation
- Variable control
- File manipulation
- #DELTA control

Many commands in #DELTA can be modified by the command flags. The command flags are the at sign (@) and the colon (:). The meaning and use of the command flags varies from command to command.

Text Manipulation Commands

Text manipulation commands work with the text in the buffer, finding, inserting, deleting, altering, and displaying the text, and moving the pointer to pinpoint where its actions are to occur. Table 9-6 summarizes the syntax and effect of the text manipulation commands. Full descriptions follow Table 9-9 on page 9-121.

Table 9-6. Text Manipulation Commands (page 1 of 3)

Command	Description	Effect on Buffer	Effect on X, Y, and Pointer (P)
*x*C	Moves characters	-	P = P + x
x:C	Moves characters with return code	-	If successful, X=-1 and P=P+X; if not, X=0
*x*D	Deletes characters	x chars deleted	-

Table 9-6. Text Manipulation Commands (page 2 of 3)

Command	Description	Effect on Buffer	Effect on X, Y, and Pointer (P)
*x*FC	Changes lines to lowercase	Lines from P to x EOLs changed to lowercase	-
y,*x*FC	Changes characters to lowercase	Chars y through x changed to lowercase	-
x@FC	Changes lines to uppercase	Lines from P to x EOLs changed to uppercase	-
y,*x*@FC	Changes characters to uppercase	Chars y through x changed to uppercase	-
I*text*$*	Inserts text	Text inserted at P	P = P + size
*x*I	Inserts ASCII character	Character inserted at P	P = P + 1
y,*x*I	Inserts y*ASCII characters	y number of characters inserted at P	P = P + y
*x*J	Jump characters	-	P = x
*x*K	Kills lines	Lines from P to x EOLs deleted	-
y,*x*K	Kills characters	Chars from y to x deleted	P = y
*x*L	Moves pointer by lines	-	P = x EOLs
*x*S*text*$*	Searches	-	P = xth occurrence
x:S*text*$*	Searches with return code	-	P = xth occurrence. If found, X=-1; if not, X=0
*x*T	Types lines	-	-
y,*x* T Types chars	-	-	
@T*var*$	Types variable level contents	-	-
:T*text*$*	Types text	-	-
*x*V	Views lines	-	-
x:V	Views lines and show ends	-	-

Table 9-6. Text Manipulation Commands (page 3 of 3)

Command	Description	Effect on Buffer	Effect on X, Y, and Pointer (P)
\	Converts number in text to value in X	-	X = (P)
x\	Puts x in text	Text value of x inserted at P	P = P + size

* You can modify the I, S, and :T commands with the @ flag to change the text delimiter from $ to another character; for example: @I/text/

Variable Control Commands

Commands in #DELTA macros can push, pop, set, and manipulate variables. All these commands (except the ^\command) must be immediately followed by a variable name; the variable name must be terminated by a dollar sign ($). Table 9-7 summarizes the syntax and effects of these commands. Full descriptions follow Table 9-9 on page 9-121.

Table 9-7. Variable Control Commands (page 1 of 2)

Command	Description	Effect On Buffer	Effect on X, Y, and Pointer (P)
FE*var*$	Tests variable level for emptiness	-	X = -1 if empty, 0 if not
FF*var*$	Gets frame number of variable level	-	X=frame number
*x*FG*var*$	Compares lines to variable level	-	If successful, X = -1; if not, X = 0
y,*x*FG*var*$	Compares range to variable level	-	If successful, X = -1; if not, X = 0
FO*var*$	Pops variable	-	-
FT*var*$	Gets var type	-	X=variable type
*x*FT*var*$	Sets var type	-	-
FU*var*$	Pushes variable	-	-
*x*FU*var*$	Pushes and loads variable with x	-	-
G*var*$	Gets text from variable level	Definition text put at P	P = P + size
M*var*$	Invokes macro	-	-
Q*var*$	Gets value from variable level	-	X = var
*x*U*var*$	Unloads x into variable level	-	-

Table 9-7. Variable Control Commands (page 2 of 2)

Command	Description	Effect On Buffer	Effect on X, Y, and Pointer (P)
y,*x*U*var*$	Unloads x into variable level	-	X = y
*x*X*var*$	Extracts lines to variable level	Lines from P to x EOLs put into variable level	-
y,*x*X*var*$	Extracts chars to variable level	Characters from y to x put into variable level	-
^\	Exits from macro	-	-

△ **Caution.** If you used the /COMMANDS/ option when you invoked #DELTA, do not push, pop, or in any other way modify the variable level from which #DELTA is receiving its commands; doing so can cause #DELTA, and possibly TACL, to fail.

File Manipulation Commands

#DELTA can open and close files and transmit data to and from them. This capability is often useful for editing a document. Table 9-8 summarizes the syntax and effects of these commands. Full descriptions follow Table 9-9 on page 9-121.

Table 9-8. File Manipulation Commands

Command Description	Description	Effect on Buffer	Effect on X, Y, and Pointer (P)
EI*file*$ *	Opens file for input	-	-
EO*file*$ *	Opens file for output	-	-
*x*P	Writes lines	Lines from P to x EOLs sent to output file	-
y,*x*P	Writes chars	Chars from y to x sent to file	-
*x*Y	Reads lines	x lines moved from file to (P)	P = P + size

* You can modify EI and EO commands with the @ flag to change the delimiter; for example: @EI/$trmnl/

#DELTA Control Commands

The #DELTA control commands are those that you can use to control the execution of #DELTA commands. The control commands govern conditional operations, iteration, and macro execution, and the placement of values in the X and Y registers to govern

command execution. Table 9-9 summarizes the syntax and effects of these commands.

Table 9-9. #DELTA Control Commands

Command	Description	Effect on Buffer	Effect on X, Y, and Pointer (P)
;	Exits iteration	-	-
?	Specifies condition	-	-
:?	Specifies NOT condition	-	-
’	Ends condition	-	-
,	Moves X into Y	-	Y=X; X cleared
$	Clears X, Y - X and Y segments and retrieves buffer position	-	X = P
=	Displays X or Y,X	-	-
<	Begins iteration	-	-
x<	Iterates x times	-	-
>	Ends iteration	-	-

The following subsections provide full descriptions of all the #DELTA commands, in alphabetic sequence.

The A Command

The A command loads the X register with a number that corresponds to the position of a character in the ASCII character set. The X register value (before execution) specifies the buffer position, relative to the pointer, of the character to be examined; 0A refers to the character before the pointer, 1A (or A) to the character after the pointer. The A command does not change the pointer position. For example:

```
57> #DELTA ARGH!
#DELTA 58> 4JV
#DELTA 58> EOF!
ARGH(.)!
#DELTA 59> 0A=
#DELTA 59> EOF!
72
#DELTA 60> 1A=
#DELTA 60> EOF!
33
```

“H” is character number 72 in the ASCII set; exclamation point is character number 33.

If the position specified by the X register is outside the text in the buffer, an error occurs. However, if you have placed a value in the Y register, #DELTA avoids the error and loads the Y register value into the X register instead.

The B Command

The B command loads the value 0 into the X register, thus indicating the beginning of the buffer.

The C Command

The C command moves the pointer the number of positions specified by the contents of the X register; you can precede the C command with an X register value. The sign of the number indicates the direction to move: forward (positive) or backward (negative). You must use a minus sign to indicate a negative number, but positive values are implied by the absence of a minus sign. If there is no value in the X register, 1 is assumed.

If you use the : flag before the C command, the command returns -1 if it succeeds, or 0 if the command would try to move the pointer outside the text in the buffer. If the command is successful, the pointer is moved the specified number of characters; if the command fails, the pointer is not moved.

For example:

```
#DELTA 28> v
#DELTA 28> EOF!
This is a test string(.)
#DELTA 29> -2:c = v
#DELTA 29> EOF!
-1
This is a test stri(.)ng
#DELTA 30> 4:c=v
#DELTA 30> EOF!
0
This is a test stri(.)ng
#DELTA 31>
```

The D Command

The D command moves the pointer a number of characters specified by the contents of the X register, deleting those characters in the process; you can precede the D command with an X register value. A minus sign preceding the number specifies that characters are to be deleted in a backward direction. You must not use a plus sign to specify forward direction, however; a positive value is implied by the absence of a minus sign. If there is no value in the X register, 1 is assumed. For example:

```
65> #DELTA argh!
#DELTA 66> 1jv
#DELTA 66> EOF!
a(.)rgh!
#DELTA 67> dht
```

```
#DELTA 67> EOF!
agh!
```

The EI Command

The EI command opens a file for input. To read a file, you must open the file and specify that the file is open for input. You cannot open the same file for both input and output, and you can have only one file open for input and one file open for output at the same time. If you try to open a nonexistent file for input, an error occurs.

If you follow the EI command with a file name, TACL closes the file currently open for input and opens the named file. If you omit the file name, TACL closes the file currently open for input. If you leave a file open when you exit from #DELTA, TACL closes it automatically.

If you specify a file name with EI, you must end the file name with a dollar sign. If the file name contains a dollar sign, you must use the @ flag with the command to specify another delimiter. The delimiter is the first character that follows the command. For example, the command @EI/$KYRIE.LEE.TESTSRC/ uses slashes (/) as its delimiters.

This example shows how you can use EI to open a file whose name is supplied as an argument:

```
#PUSH fn opfl
#SET fn %1%                          == Save file name in fn
#SET /TYPE DELTA/ opfl @EI/[fn]/ == Build custom EI command
[#DEF delcomm DELTA |BODY|       == Define DELTA commands
 ...
Mopfl$                               == Invoke custom EI command
 ...
]
#DELTA /COMMANDS delcomm/       == Invoke DELTA commands
```

The EO Command

The EO command opens a file for output. To write to a file, you must open the file and specify that the file is open for output. You cannot open the same file for both input and output, and you can have only one file open for input and one file open for output at the same time. If a file opened for output does not exist, TACL creates an edit file with that name; if the file does exist, #DELTA appends the lines it writes to the end of the file.

If you follow the EO command with a file name, TACL closes the file currently open for output and opens the named file. If you omit the file name, #DELTA closes the file currently open for output. If you leave a file open when you exit from #DELTA, TACL closes it automatically.

If you specify a file name with EO, you must end the file name with a dollar sign. If the file name contains a dollar sign, you must use the @ flag with the command to specify another delimiter. The delimiter is the first character that follows the command. For example, the command @EO'$FURD.UFFDA.TESTSRC' uses apostrophes (') as its delimiters.

The FC Command

The FC command changes the case of text within a specified range. You can specify the range of text either with an X register value (X lines of text, starting at the current pointer position), or with X and Y register values (from character Y to character X). If you use the @ flag with the FC command, the text in the range is changed to uppercase; if you do not use the @ flag, the text is changed to lowercase.

If FC appears at the beginning of an interactive command line, it is the standard FC (Fix Command) command.

The FE Command

The FE examines a variable level to see if it is empty. If the variable level contains nothing, FE loads -1 into the X register; if the variable level contains anything, FE loads 0 into the X register. FE considers a variable level to be empty only if it does not contain anything, even a space or an end-of-line.

The FF Command

The FF command loads the X register with the frame number associated with a specified variable level. The FF command must be followed by a variable level name. The variable level name must be terminated by a dollar sign.

The FG Command

The FG command compares a variable level with a specified range of text. You can specify the range of text either with an X register value or with X and Y register values. The comparison is not case-sensitive. If the comparison succeeds, FG loads -1 into the X register; if the comparison fails, it loads 0 into the X register.

The comparison succeeds only if all characters in the variable level agree with the buffer text. In this example, the comparison fails because DIGITS contains 10 characters (the 8 digits assigned plus a two-byte end-of-line character) and only 9 characters in the buffer are compared with it:

```
70> #PUSH digits
71> #SET digits 12345678
72> #DELTA
#DELTA 73> Gdigits$ BJ .,.+8FGdigits$ =
#DELTA 73> EOF!
0,0
```

If you had used this command instead:

```
#DELTA 73> Gdigits$ BJ HFGdigits$
```

the comparison would have succeeded because the entire contents of DIGITS were in the buffer (the end-of-line character is represented by a null character).

The FL Command

The FL command loads the X register with the length of the string last inserted by I or found by S. This is especially useful for doing text replacement where you do not want to (or cannot) count the length of the search string. For example, you might use the following in a TACL macro because the length of the first argument of the macro varies from call to call:

```
#SET /TYPE DELTA/ cmds ... J <@:S/%1%/; -FL D> ...
#DELTA / COMMANDS cmds /
```

The iteration searches the buffer for a string that matches an argument of unknown length; if it finds the string, it sets the X register to the negative (-FL) value of the string length and deletes that many characters from the buffer. FL is set to zero whenever #DELTA prompts.

The FO Command

The FO command pops a TACL variable. If you pop the last level of a variable, that variable is destroyed.

The FT Command

You use the FT command to get or set the type of a variable level. FT must be followed by the name of a variable level. If the X register is clear, FT returns a value representing the variable type in the X register.

The values and their types are:

Value	Type
1	MACRO
2	ROUTINE
3	TEXT
4	DELTA
5	ALIAS
6	DIRECTORY
7	STRUCT

If the X register contains a value between 1 and 6, FT sets the specified variable level to the type represented by the value.

The FU Command

The FU command pushes a TACL variable. If you push a variable that does not exist, #DELTA creates it. If there is a value in the X register when you use the push command, the new level of the variable contains that value.

The G Command

The G command copies text from a variable level and inserts it in the buffer, moving the pointer to the right of the text. The G command must be followed by a variable level name; that name must be terminated by a dollar sign.

If you precede the G command with the : flag, #DELTA extracts (and deletes) the first line from the variable level and puts it in the buffer. Without the : flag, #DELTA copies the entire variable level into the buffer and the variable level remains unchanged.

The X command is the complementary function of the G command.

The H Command

The H command is equivalent to using the commands B,Z; that is, it loads the Y register with the beginning of the buffer and the X register with the size of the buffer. You can use the H command in conjunction with other commands to perform an operation on the whole buffer. For example, the command HT displays the entire buffer.

The I Command

The I command inserts text into the buffer at the current position. There are three different ways to use the I command:

- If there is no value in either X or Y registers and the I command is followed by a text string terminated by a dollar sign, #DELTA inserts that text.
- If there is a value in the X register, #DELTA inserts the ASCII character that corresponds to that value into the buffer.
- If there are values in the X and Y registers, #DELTA inserts the ASCII character that corresponds to the X register into the buffer Y number of times.

This example illustrates the three different uses of the I command:

```
41> #DELTA test
#DELTA 42> j iA $
#DELTA 42> 1j v
#DELTA 42> EOF!
A(.) test
#DELTA 43> 32i v !32 is ASCII code for space
#DELTA 43> EOF!
A (.) test
#DELTA 44> 5,33i v !33 is ASCII code for exclamation point
#DELTA 44> EOF!
A !!!!!(.) test
#DELTA 45>
```

Only when there is no value in either X or Y register can you follow the I command with text; in the other two cases, the I command must not be followed by text. If the text string contains a dollar sign, you can use the @ flag with the I command to specify new

delimiters around the string; the character immediately to the right of the I command is the new delimiter. The delimiter is changed only for the current I command.

For example:

```
#DELTA 45> @I/new$/ v
#DELTA 45> EOF!
a !!!!!new$(.) test
#DELTA 46> I string$ v
#DELTA 46> EOF!
a !!!!!new$ string(.) test
#DELTA 47>
```

#DELTA uses the ASCII code 0 to mean end-of-line; thus, 0I (or I) breaks a line.

The J Command

The J command moves the pointer to an absolute location specified by the contents of the X register. Thus, 0J jumps to the beginning of the buffer; 23J jumps to the 23rd character in the buffer. The J command accepts an X register value; if none is specified, 0 is assumed.

The K Command

Use the K command to delete a specified range of text, in lines or characters. The range of text can be specified either with an X register value or with X and Y register values.

If you specify only an X register value, K deletes x lines starting at the current position. (If x is negative, K deletes x lines preceding the current position.) The K command with no range is equivalent to 1K; -K is equivalent to -1K.

If you specify both an X register value and a Y register value, K deletes a range of characters from position y to position x.

The difference between the K and D commands is that K deletes lines or any range of characters; D deletes only characters, starting at the current position.

The L Command

The L command moves the pointer the number of lines specified by the contents of the X register. That value indicates the number of end-of-line characters to skip in search of a new line. The command 0L moves to the beginning of the current line; 1L moves to the beginning of the next line. The L command accepts an X register value; if none is specified, 1 is assumed; -L is equivalent to -1L.

The M Command

The M command executes a #DELTA macro. The M command must be followed by a variable level name, which must be type DELTA. The name must be terminated by a dollar sign. The macro uses the same buffer and current pointer value as the command stream that invoked it. The commands in the macro are executed until the macro

encounters the ^\command or the end of the commands in the variable level. When the macro exits, the buffer and current pointer value remain where the macro left them.

The only way to pass arguments to #DELTA macros is to load the arguments into variable levels before calling the macro, then, from within the macro, get the contents of the variables.

The P Command

The P command writes a range of text to the output file. The range of text can be specified either with an X register value (X lines of text, starting at the current pointer position) or with X and Y register values (character Y through character X). You cannot represent a partial line in a file. If you output a line fragment, that fragment becomes a new line in the output file.

#DELTA does all file I/O in PLAIN mode.

The Q Command

The Q command moves a value from a numeric variable level into the X register. The variable level name must be terminated by a dollar sign.

To move a numeric variable level into text, you can either insert the variable level directly with the G command or move the value into the X register with Q and then insert the contents of the X register into text with the backslash command (that is, Qvar$\).

The U command is the complementary function of the Q command.

The S Command

The S command searches the buffer for a specified string, starting at the current position. The search is not case-sensitive. The contents of the X register specify the search direction-a positive value means search forward; a negative value, backward-and how many occurrences of the string to search for before stopping. The string to be sought follows the S command and is normally terminated by a dollar sign. If the search string contains a dollar sign, you can use the @ flag to specify a different delimiter; for example:

```
54> #DELTA This is a $test string
#DELTA 55> bj @s/$test/ v
#DELTA 55> EOF!
This is a $test(.) string
#DELTA 56>
```

In this example, the string “$test” contains a dollar sign, so you use the @ flag to specify a slash (/) as the delimiter. The new delimiter must follow the S command. Using the @ flag changes the delimiter for only the S command that it modifies.

If the search direction is forward, the pointer is set to the right of the matching string; if the search is backward, the pointer is set to the left of the match.

If the search string is not found, #DELTA issues an error message and, if the #DELTA session is not interactive, #DELTA exits. The pointer does not move. You can use the : flag to enable your #DELTA code to handle search errors-if the string is found, the X register is set to -1; if not, it is set to 0.

You can also use S with FL to do text replacement.

The T Command

The T command displays a range of text, the contents of a variable level, or a user-specified string.

The range of text can be specified either with an X register value or with X and Y register values.

To display the contents of a variable level, use the @ flag with the T command. The T command must be immediately followed by the variable level name; the variable level name must be terminated by a dollar sign.

For example:

```
60> #PUSH TEST
61> #SET TEST Test string in a variable
62> #DELTA
#DELTA 63> @Ttest$
#DELTA 63> EOF!
Test string in a variable
#DELTA 64>
```

You can use the : flag to direct the T command to display a text string. The text follows the T command and must be terminated with a dollar sign. If the text contains a dollar sign, you can use the @ flag to specify the string delimiter. The delimiter is the first character after the T command. In this example, the first :T command uses the dollar sign terminator; the second :T command types out a string that contains a dollar sign, so it uses the @ flag to specify a different terminator:

```
#DELTA 64> :TParsing string...$
#DELTA 64> EOF!
Parsing string...
#DELTA 65> @:T/Couldn't open $GERT.STEIN.NEWMACS/
#DELTA 65> EOF!
Couldn't open $GERT.STEIN.NEWMACS
#DELTA 66>
```

The U Command

The U command moves the value in the X register into a numeric variable level, consisting of one line, the text of which is the ASCII representation of a number. The variable level name must be terminated by a dollar sign.

The U command stores the value of the X register in the variable level, stores the Y register value in the X register, and sets the Y register to a null value. A second U

command could then save the former Y register value in another variable level. For example:

```
123,456 Uxval$ Uyval$
```

puts 456 into XVAL, 123 into YVAL, and clears the X and Y registers.

The U command is the complementary function of the Q command.

The V Command

The V (view) command is most commonly used as an aid in debugging #DELTA macros and, in general, finding out where you are in the buffer.

The V command displays lines of text and indicates where the pointer is. The X value determines the number of lines displayed by the V command. 1V (or V) displays the line that includes the current position; 2V displays the line that includes the current position, plus one line in either direction.

The number of lines to display is actually determined by the number of end-of-line characters encountered in either direction from the current position. Note that end-of-line characters include the ends-of-lines at either end of the current line. If there is no text in either direction, the V command does not report an error.

If you use the : flag with the V command, #DELTA indicates the beginning (B) or end (Z) of the buffer, if they are within the range specified with the V command.

The X Command

The X command loads a TACL variable level with a range of text in the buffer. The X command must be immediately followed by the name of a variable level; the variable level name must be terminated by a dollar sign.

The range of text can be specified either with an X register value or with X and Y register values: *x*X *var-name*$ loads *x* number of lines from the buffer into *var-name*; *y, x*X v*var-name*$ loads characters from positions *y* through *x* into *var-name*.

If you precede the X command with the : flag, #DELTA appends the text to that already in the variable level. Without the colon flag, #DELTA replaces any existing text in the variable level with text from the buffer.

The Y Command

The Y command reads text from the input file into the buffer. The contents of the X register indicate the number of lines to read. If there is no value in the X register, #DELTA reads, or tries to read, all the lines in the file into the buffer (remember that the buffer is less than 30,000 characters long).

When the Y command succeeds in reading text into the buffer, it loads the number of lines read into the X register. If the Y command reads past the end of file, it closes the

file. If you use the Y command on a closed file, it loads the value zero into the X register.

#DELTA does all file I/O in PLAIN mode.

The Z Command

The Z command loads the X register with the size of the currently used portion of the buffer (in characters). You can use the Z command as a pointer to the end of the buffer. For example, the commands Z K delete the characters from the current position to the end of the buffer.

The $ Command

The dollar sign command clears both the X and Y registers.

You may want to use the dollar sign command before a command that can behave differently depending on whether there are values in the X or Y registers.

The end-of-line following a line of #DELTA commands also clears the X and Y registers.

The , Command

The comma command moves the contents of the X register to the Y register and clears the X register. You can then load a second value into the X register. For example:

```
#DELTA 73> 8 =
#DELTA 73> EOF!
8
#DELTA 74> 8, =

#DELTA 74> EOF!
8,0
#DELTA 75> 8,40 =
#DELTA 75> EOF!
8,40
#DELTA 76>
```

If you enter another comma and another new X register value, the first Y register value is lost.

The . Command

The period command loads the current pointer position into the X register.

The = Command

The equal sign command displays the contents of the X and Y registers on your current output device. The = command is useful for debugging macros. If you are not sure what value is in the X and Y registers before a command is executed, insert the equal sign before that command. (Remember to remove the equal sign when you have finished debugging; the = command destroys the contents of the X and Y registers.)

The \ Command

The backslash command converts a value in the X register to a number in text and vice versa.

If there is no value in the X register and the text to the right of the pointer is a number,

#DELTA converts the text to a numeric value and loads it into the X register. For example, if the text is 123ABC, the \ command loads 123 into the X register. If the text is nonnumeric or is empty, an error occurs.

If there is a value in the X register, #DELTA inserts the textual representation of the value into the buffer at the current position, moving the pointer to the right of the inserted number.

If there is a value in the Y register, #DELTA inserts the value from the X register (zero, if there is none) and right-justifies the inserted number in a field y characters wide. If y is zero or greater, #DELTA fills the field with leading spaces; if y is less than zero, leading zeros are used instead. If the absolute value of y is less than or equal to the number of digits in the value to be inserted, #DELTA simply inserts it.

The ^\ Command

This command terminates a #DELTA macro begun by the M command.

Conditional Processing (? and ’ Commands)

Conditional processing is controlled by a construct that begins with a question mark (?) followed by a letter indicating the condition to be tested for, then the command(s) to be conditionally executed, and ends with an apostrophe (’). All tests are done on the contents of the X register. If the test is successful, #DELTA executes the command or commands between the test letter and the apostrophe. The test letters are:

Letter	Condition to be Tested
A	Is X the ASCII value of an alphabetic character?
D	Is X the ASCII value of a numeric character?
E	Is X equal to zero?
G	Is X greater than zero?
L	Is X less than zero?
N	Is X not equal to zero?

The : flag, placed before the question mark, is the NOT operator. For example, 1?E is false, but 1:?E is true.

In this example, the character before the pointer is converted into its ASCII value and is loaded into the X register. The conditional test then tests whether the character is an alphabetic character. If it is not, #DELTA replaces it with a space (-D I $ deletes the character and inserts a space):

```
0A :?A -D I $'
```

Conditional constructs can be nested.

Iteration (< and >Commands)

Iterations (groups of #DELTA commands to be performed repeatedly) are delimited by angle brackets. If there is a value in the X register, #DELTA performs the iteration that number of times. If there is no value in the X register, #DELTA repeats the iteration until you explicitly end it from within the loop. Each time #DELTA repeats the iteration, it decrements a counter; if, however, you use the @ flag before the closing greater-than sign (@>), the iteration count is not decremented.

You can put a conditional construct containing an end-iteration command (>or @>) within the iteration: If the condition is true, the end-iteration command forces the loop to start its next execution before reaching the actual end of the iteration.

You can use the semicolon command (;) within an iteration to force its termination, based on a test of the value in the X register:

; Exit if X register is greater than or equal to zero.

@; Exit if X register is equal to zero.

:; Exit if X register is less than zero.

@:; Exit if X register is not equal to zero.

You can nest iterations.

The following iterative statement searches for all occurrences of the British spelling of the word color (colour), and changes it to the American spelling by deleting the letter “u.” The : flag before the S command directs the search to return a status value. If the value is zero, the search failed and the semicolon command ends the iteration:

```
bj <:Scolour$;-c -d>
```

#DEVICEINFO Built-In Function

Use #DEVICEINFO to obtain information about a device.

```
#DEVICEINFO / option [ , option ] ... /
  { device-name | file-name }
```

option

identifies the specific information you are requesting. It must be one of these:

AUDITED

requests the value of the AUDITED flag of the specified device or file. If the AUDITED flag is set for a device, files audited by the TMF subsystem can reside on the device. If the AUDITED flag is set for a file, the file is audited by TMF. For more information, see the *TMF Reference Manual*.

DEVICETYPE

requests the device type.

RECORDLENGTH

requests the device physical record length, in bytes.

SUBTYPE

requests the device subtype.

Device type and subtype are represented as decimal integers; see the *Guardian Procedure Calls Reference Manual* for the type and subtype values.

device-name

is the syntactically correct name of a device to be examined (or the process that drives it).

file-name

is the syntactically correct name of a file stored on the device to be examined.

Result

#DEVICEINFO returns the information requested by the options. If you specify more than one option, #DEVICEINFO lists the items of information, separated by spaces, in the same order as the option requests. If you specify a device or file that does not exist, or if an error occurs, #DEVICEINFO returns the following:

- 0 for AUDITED, DEVICETYPE, and SUBTYPE
- 132 for RECORDLENGTH

#EMPTY Built-In Function

Use #EMPTY to determine if some specified text is empty.

```
#EMPTY [ text ]
```

text

is the text to be examined.

Result

#EMPTY returns -1 if text is empty; otherwise, it returns 0.

Considerations

- TACL considers text to be empty if it contains nothing but spaces or end-of-lines.
- Use the #EMPTYV built-in function to determine whether a variable level or quoted text is empty.

Example

This example from a macro verifies that the nth argument contains text before processing that argument.

```
[#IF NOT [#EMPTY % n%] |THEN|
 ...
  code for nonempty nth argument
 ...
]
```

#EMPTYV Built-In Function

Use #EMPTYV to determine whether a variable level or quoted text is empty.

```
#EMPTYV [ / BLANK / ] string
```

BLANK

specifies that a string containing only spaces or end-of-lines is to be considered empty.

string

is the name of an existing variable level, text enclosed in quotation marks, or a concatenation of such entities. The concatenation operator is '+' (the apostrophes are required).

Result

#EMPTYV returns -1 if the string is empty; otherwise, it returns 0.

Consideration

Use the #EMPTY built-in function to determine if specified text is empty.

#EMSADDSUBJECT Built-In Function

Use #EMSADDSUBJECT to add a subject token to the event message buffer. This function places the token that you provide, preceded by a subject-mark token, in the buffer.

```
#EMSADDSUBJECT [ / SSID ssid / ] buffer-var
  token-id [ token-value ]
```

`SSID` *`ssid`*

is a subsystem ID that qualifies the token code; if omitted or zero (0.0.0), it defaults to the subsystem ID of the current list or, if the current position is not in a list, to the subsystem ID specified in the event message header (ZSPI-TKN-SSID).

`buffer-var`

is the name of a writable STRUCT used as an EMS buffer. #EMSADDSUBJECT automatically passes the data length of the STRUCT to the EMSADDSUBJECT system procedure.

`token-id`

is the token code of the subject token to be added to the event message.

`token-value`

is a value to be given to the new token. Include this parameter if a value is associated with the token.

Result

#EMSADDSUBJECT returns a numeric status code indicating the outcome of the EMSADDSUBJECT procedure.

The meaning of the status code:

Code	Condition
0	No error
-1	Invalid buffer format
-2	Invalid parameter value
-3	Missing parameter
-4	Invalid parameter address
-5	Buffer full
-6	Invalid checksum
-7	Internal error
-8	Token not found

Code	Condition
-9	Invalid token code or map
-10	Invalid subsystem ID
-11	Operation not supported
-12	Insufficient stack space

Considerations

- Every event message has at least one subject, which you specify to the #EMSINIT procedure. Use #EMSADDSUBJECT to specify additional subjects.
- #EMSADDSUBJECT inserts two tokens into the buffer: the ZEMS-TKN-SUBJECT-MARK token, which always precedes a subject and the subject token you specified.
- If the subsystem that generates the event message needs to include tokens from another subsystem (often a lower-level subsystem), its call to #SSPUT(V) must include the SSID option, which specifies the subsystem ID of the other subsystem. When you specify the SSID option, every token placed in the buffer on that procedure call is in an extended form that includes the SSID you specified.
- To supply token values from a variable level, use the #EMSADDSUBJECTV built-in function.

#EMSADDSUBJECTV Built-In Function

Use #EMSADDSUBJECTV to add a subject token to the event message buffer. This function places binary token values taken from a variable level, preceded by a subject-mark token, in the buffer.

```
#EMSADDSUBJECTV [ / SSID ssid / ] buffer-var
  token-id source-var
```

`SSID` *`ssid`*

is a subsystem ID that qualifies the token code; if omitted or zero (0.0.0), it defaults to the subsystem ID of the current list or, if the current position is not in a list, to the subsystem ID specified in the event message header (ZSPI-TKN-SSID).

`buffer-var`

is the name of a writable STRUCT used as an EMS buffer. #EMSADDSUBJECTV automatically passes the data length of the STRUCT to the EMSADDSUBJECT system procedure.

`token-id`

is the token code or token map of the subject token to be added to the event message.

`source-var`

is the name of a variable level, of type STRUCT, from which #EMSADDSUBJECTV is to obtain binary token values. The contents of the STRUCT are not altered.

Result

#EMSADDSUBJECTV returns a numeric status code indicating the outcome of the EMSADDSUBJECT procedure.

The meaning of the status code:

Code	Condition
0	No error
-1	Invalid buffer format
-2	Invalid parameter value
-3	Missing parameter
-4	Invalid parameter address
-5	Buffer full
-6	Invalid checksum
-7	Internal error

Code	Condition
-8	Token not found
-9	Invalid token code or map
-10	Invalid subsystem ID
-11	Operation not supported
-12	Insufficient stack space

Considerations

- Every event message has at least one subject, which you specify to the #EMSINIT(V) procedure. Use #EMSADDSUBJECTV to specify additional subjects.
- #EMSADDSUBJECTV inserts two tokens into the buffer: the ZEMS-TKN-SUBJECT- MARK token, which always precedes a subject, and the subject token variable or the STRUCT.
- If the subsystem that generates the event message needs to include tokens from another subsystem (often a lower-level subsystem), its call to #SSPUT(V) must include the SSID option, which specifies the subsystem ID of the other subsystem. When you specify the SSID option, every token placed in the buffer on that procedure call is in an extended form that includes the SSID you specified.

#EMSGET Built-In Function

Use #EMSGET to retrieve binary token values from an SPI buffer, convert them to external representation, and make that external representation accessible in the result of the function. #EMSGET invokes the EMSGET operating system procedure.

You cannot use #EMSGET to extract values of extensible structured tokens using a token map or using a token code of type ZSPI^TDT^STRUCT. Instead, use #EMSGETV.

```
#EMSGET [ / option [ , option ] ... / ] buffer-var get-op
```

option

is any of these:

COUNT *count*

is the maximum number of token values to be returned. This specifies that the token value returned is an array of count elements, each of which is described by *token-code*. If you omit count, it defaults to 1.

If count is greater than 1, #EMSGET continues searching until it either satisfies the requested count or reaches the end of the buffer or list.

If count is less than 1, an error occurs.

INDEX index

is the specific occurrence of *token-code*, starting from the beginning of the buffer or list (an index of 1 gets the first occurrence of that token code, an index of 2, the second, and so on).

If you omit this option or if index = 0, #EMSGET returns the next occurrence of the token code after the current position, and resets the current position to that of the token value returned. If you want to search from the beginning of the buffer, you must supply a nonzero index or else have previously reset the initial position with #SSPUT(V) using ZSPI^TKN^INITIAL^POSITION or ZSPI^TKN^RESET^BUFFER.

If index is less than zero, an error occurs.

SSID *ssid*

is a subsystem ID that qualifies the token code; if omitted or zero (0.0.0), it defaults to the subsystem ID of the current list or, if the current position is not in a list, to the subsystem ID specified in the SPI message header. The version field of this parameter is not used when searching the buffer.

buffer-var

is the name of the message buffer from which information is to be taken; it must be a writable STRUCT that has been initialized by #SSINIT.

get-op

is one of these:

token-code

directs #EMSGET to return the token value or values associated with *token-code*.

If *token-code* is a token that marks the beginning of a list, #EMSGET selects the list so that subsequent calls can retrieve tokens within the list. If *token-code* is ZSPI^TKN^ENDLIST, #EMSGET pops out of the list.

token-code can be any of the header tokens listed under “Header Tokens and Special Operation for #SSGET and #SSGETV” in the description of the #SSGET built-in function. You can also supply the token ZSPI^TKN^DEFAULT^SSID to obtain the default subsystem ID at the current position.

ZSPI^TKN^COUNT *c-token-id*

directs #EMSGET to return the number of occurrences of the token specified by the token code or the token map *c-token-id*, starting with the occurrence specified by index. To count all occurrences in the list, specify an index of 1.

If *c-token-id* is omitted or equal to ZSPI^VAL^NULL^TOKENCODE, and index is omitted or zero, #EMSGET counts occurrences of the current token, including the current occurrence of that token.

ZSPI^TKN^LEN *l-token-id*

directs #EMSGET to return the byte length of the token specified by the token code or token map *l-token-id*. This is the size of the buffer needed to contain the stated occurrence of the token value. For variable-length token values, this includes the two bytes required for the length word: The byte length returned is *token-value*[0]+2.

If this option is omitted or *l-token-id* is ZSPI^VAL^NULL^TOKENCODE, and index is omitted or zero, #EMSGET returns the length of the current occurrence of the current token.

If *l-token-id* is a token map, this operation returns the length contained in that map; the value in the buffer can be longer or shorter than this length. To get the actual length of the token value, call #EMSGET with ZSPI^TKN^LEN and a token code made up of ZSPI^TYP^STRUCT and the token number from the token map. This returns the length of the value, including 2 bytes for the length field. Subtract 2 from this result to get the length of the value itself.

ZSPI^TKN^NEXTCODE

directs #EMSGET to return the next token code that is different from the current token code, followed by the subsystem ID. The subsystem ID returned in the result always has a version field of zero (null).

The index parameter has no effect on this operation, but if you supply it, it must be zero.

ZSPI^TKN^NEXTTOKEN

directs #EMSGET to return the next token code, followed by the subsystem ID. The subsystem ID returned always has a version field of zero (null).

This operation differs from ZSPI^TKN^NEXTCODE in that it always returns the token code of the next token, whether it is the same as that of the current token or different, and whether the token is within a list or not. The operation returns multiple occurrences of the same token code in the same order as they were added to the buffer with #SSPUT(V).

The index and count parameters have no effect on this operation, but if you use them, index must be zero; count is always returned as 1.

Note. The special operations ZSPI^TKN^NEXTCODE and ZSPI^TKN^NEXTTOKEN return only token codes. In particular, note that tokens added to the buffer using #SSPUTV with a token map are carried in the buffer with a token code of type ZSPI^TYP^STRUCT. The NEXTCODE and NEXTTOKEN operations return that token code, not the token map used with #SSPUTV.

ZSPI^TKN^OFFSET *o-token-id*

directs #EMSGET to return the byte offset of the token specified by the token code or token map *o-token-id*. The value returned is the offset from the start of the buffer to the value associated with the specified token code and index. (For variable-length values, the token value begins with the length word; the offset given is the offset to that length word.)

If you omit this option or if *o-token-id* is equal to ZSPI^VAL^NULL^TOKENCODE, and index is omitted or zero, #EMSGET returns the length of the current occurrence of the current token.

You must supply appropriate token code definitions. TACL supports the special semantics for only those SPI special token codes shown; any other token codes are assumed to adhere to standard semantics.

Result

#EMSGET returns a numeric status code indicating the outcome of the SSGET procedure.

The meaning of the status code:

Code	Description
0	No error
-1	Invalid buffer format
-2	Invalid parameter value
-3	Missing parameter
-4	Invalid parameter address
-5	Buffer full
-6	Invalid checksum
-7	Internal error
-8	Token not found
-9	Invalid token code or map
-10	Invalid subsystem ID
-11	Operation not supported
-12	Insufficient stack space

If the status code is 0 (no error), it is followed by a space and a space-separated list of the relevant EMSGET results in the TACL external representation:

- If you specified *token-code*, #EMSGET returns the number of token values returned, followed by a space-separated list of those values in external form; variable-length token values are returned in two parts-the byte length, followed by the actual value-separated by a space.
- If you specified ZSPI^TKN^COUNT *c-token-id*, #EMSGET returns the number of occurrences of the specified token, starting at the occurrence specified by index.
- If you specified ZSPI^TKN^LEN *l-token-id*, #EMSGET returns the length of the token specified by the token code or token map *l-token-id*.
- If you specified ZSPI^TKN^NEXTCODE, #EMSGET returns he next token code that is different from the current token code, followed by the subsystem ID.
- If you specified ZSPI^TKN^NEXTTOKEN, #EMSGET returns the next token code, regardless of whether it is different from the current token code, followed by the subsystem ID.
- If you specified ZSPI^TKN^OFFSET *o-token-id*, #EMSGET returns the byte offset of the token specified by *o-token-id*.

Considerations

- Tokens extracted by #EMSGET are not deleted or removed from the buffer.

- When the current position is within a particular list, all #EMSGET calls pertain only to tokens within that list (except that header fields are always accessible). You can exit from the list by calling #EMSGET to get the ZSPI^TKN^ENDLIST token.
- When *token-code* is ZSPI^TKN^ENDLIST, the index and count parameters have no effect. However, if you supply them, index must be 0 or 1; count is always returned as 1.
- To retrieve token values into a STRUCT, use the #EMSGETV built-in function.

#EMSGETV Built-In Function

Use #EMSGETV to obtain binary token values from an SPI buffer and put them into a STRUCT. You can use #EMSGETV with any type of token (in fact, you must use #EMSGETV with extensible structured tokens and tokens of type ZSPI^TYP^STRUCT). #EMSGETV invokes the EMSGET procedure.

```
#EMSGETV [ / option [ , option ] ... / ] buffer-var get-op
  result-var
```

option

is any of these:

```
COUNT count
INDEX index
SSID ssid
```

These are the same as those described for #EMSGET, substituting *token-id* for all references to *token-code*.

buffer-var

is the same as described for #EMSGET.

get-op

is one of these:

token-id

is either a token code or a token map. It directs #EMSGETV to return the token value or values associated with *token-id*.

If *token-id* is a token that marks the beginning of a list, #EMSGETV selects the list so that subsequent calls can retrieve tokens within the list. If *token-code* is ZSPI^TKN^ENDLIST, #EMSGETV pops out of the list.

```
ZSPI^TKN^COUNT c-token-id
ZSPI^TKN^LEN l-token-id
ZSPI^TKN^NEXTCODE
ZSPI^TKN^NEXTTOKEN
ZSPI^TKN^OFFSET o-token-id
```

are the same as those defined for #EMSGET, except that #EMSGETV returns results in *result-var*, rather than in the function result.

You must supply appropriate token code definitions. TACL supports the special semantics for only those SPI special token codes shown; any other token codes are assumed to adhere to standard semantics.

result-var

is the name of the writable STRUCT into which #EMSGETV is to store the data returned. The original contents of the STRUCT are lost.

If the status code in the function result is zero (no error), the result stored in *result-var* is:

- If you specified *token-id*, the result is the value of the token.
- If you specified ZSPI^TKN^COUNT *c-token-id*, the result is an INT giving the number of occurrences of the token specified by the token code or token map *c-token-id*, starting at the position specified by index.
- If you specified ZSPI^TKN^LEN *l-token-id*, the result is an INT giving the length of the specified token.
- If you specified ZSPI^TKN^NEXTCODE, the result is an INT2 giving the next token code that is different from the current token code, an INT giving the number of contiguous occurrences of that token code, and the subsystem ID.
- If you specified ZSPI^TKN^NEXTTOKEN, the result is an INT2 giving the next token code, whether different from or identical to the current token code, followed by the subsystem ID.
- If you specified ZSPI^TKN^OFFSET *o-token-id*, the result is an INT2 giving the byte offset of the token specified by *o-token-id*.

Result

#EMSGETV returns a numeric status code indicating the outcome of the SSGET procedure. The meaning of the status code:

Code	Description
0	No error
-1	Invalid buffer format
-2	Invalid parameter value
-3	Missing parameter
-4	Invalid parameter address
-5	Buffer full
-6	Invalid checksum
-7	Internal error
-8	Token not found
-9	Invalid token code or map
-10	Invalid subsystem ID
-11	Operation not supported
-12	Insufficient stack space

If no error occurred, and *get-op* is *token-id*, the status code is followed by a space and the number of token values returned.

Considerations

- Tokens extracted by #EMSGETV are not deleted or removed from the buffer.
- When the current position is within a list, all #EMSGETV calls pertain only to tokens within that list (except that header fields are always accessible). You can exit from the list by calling #EMSGETV to get the ZSPI^TKN^ENDLIST token.
- When *token-id* is ZSPI^TKN^ENDLIST, the index and count parameters have no effect; however, if you supply them, index must be 0 or 1.
- When using #EMSGETV with a token map for the *token-id* parameter, the map can specify a structure version that is longer or shorter than the structure contained in the buffer. If the requested version is longer than the version in the buffer, #EMSGETV calls SSNULL to set to null values the new fields that are not obtained from the buffer. If the requested version is shorter than the one in the buffer, #EMSGETV returns only the requested length.
- If the data returned by #EMSGETV is longer than the data area of the STRUCT identified by *result-var*, TACL discards the excess bytes without notification. If the data is shorter than the data area of *result-var*, TACL sets the entire STRUCT to its default values, then overwrites the beginning of the data bytes of the STRUCT with the returned data. No type conversions of any kind are done. This means that, for instance, if the token retrieved is of type ZSPI^TYP^INT and the *result-var* STRUCT consists of a single field of type INT2, the token value would be in the high-order 16 bits of the INT2 field, not the low-order 16 bits.
- If you specified the COUNT option, TACL puts all occurrences of the token value into the STRUCT exactly as returned by #EMSGETV, subject to the size constraints mentioned in the previous consideration. If the tokens are variable-length tokens, each token value consists of a length word followed by the actual value, and the actual value is word-aligned.
- To retrieve tokens and convert them to external representation as the result of the function call, use the #SSGET built-in function.
- Header tokens, and one special operation, can be passed in *token-id* to get corresponding values. They are described under “Header Tokens and Special Operation for #SSGET and #SSGETV” in the explanation of the #SSGET built-in function.

Example

To extract a subject token, follow two steps:

1. Call #EMSGET to get the subject token:

   ```
   #SETMANY spi^err _ subject^token [#EMSGET /INDEX 1/ &
   event^buf zems^subject^token
   ```

2. Call #EMSGETV to get the value:

   ```
   #SETMANY spi^err _, [#EMSGETV /INDEX 1/ &
   event^buf subject^token subject^value]
   ```

#EMSINIT Built-In Function

Use #EMSINIT to initialize a STRUCT as an event message buffer, preparing it for use with the other #EMS*xxx* and #SS*xxx* built-in functions. This operation gives the buffer an appropriate header and adds event information to the buffer.

```
#EMSINIT [ / option [ , option ] / ] buffer-var ssid
  event-number token-id [ token-value ]
```

option

is either of these:

SSID *ssid*

is the subsystem ID of the subsystem to which the subject token belongs. If omitted or zero (0.0.0), it defaults to the SSID of the subsystem generating the event.

TIMESTAMP *timestamp*

is the timestamp, in external representation of four-word Greenwich mean time format, for the event message. If you omit this parameter, the current time is used.

buffer-var

is the name of a writable STRUCT to be used as an EMS buffer. #EMSINIT automatically passes the data length of the STRUCT to the EMSINIT procedure. Any current contents of the STRUCT are lost.

ssid

is the subsystem ID of the subsystem generating the event.

event-number

is a number, specific to this subsystem, that identifies this event message.

token-id

is the token code of the subject of this event message.

token-value

is the token value in external representation. Its data representation is determined by the token-type field of *token-id*.

Result

#EMSINIT returns a numeric status code indicating the outcome of the initialization. The meaning of the code:

Code	Description
0	No error
-2	Invalid parameter value
-3	Missing parameter
-4	Invalid parameter address
-5	Buffer full
-7	Internal error
-10	Invalid subsystem ID
-12	Insufficient stack space
-30	Buffer length larger than ZEMS-VAL-EVT-BUFLEN

Considerations

- You can omit the token-value parameter if the token length specified by token ID is zero. Otherwise, the token-value parameter is required.
- To initialize a buffer and supply event information from a variable, use the #EMSINITV built-in function.
- Buffer length larger than ZEMS-VAL-EVT-BUFLEN indicates that the length of the STRUCT passed as the *buffer-var* parameter to the #EMSINIT built in exceeds ZEMS-VAL-EVT-BUFLEN (whose current value is 4024 bytes).

Example

Here is an example of an #EMSINIT call, using the TMF subsystem:

```
#PUSH ems^err event^num
[#DEF buf STRUCT
  BEGIN
    INT length;
    CHAR text (0:255);
  END;
]

#SET event^num 1
#SET ems^err &
  [#EMSINIT /SSID/ ZTMF^VAL^SSID TIMESTAMP &
    [#JULIANTIMESTAMP 2] &
    buf [ZTMF^VAL^SSID] [event^num] ZEMS^CMD^CONTROL]
```

#EMSINITV Built-In Function

Use #EMSINITV to initialize a STRUCT as an event message buffer, preparing it for use with the other #EMS*xxx* and #SS*xxx* built-in functions. This operation gives the buffer an appropriate header and adds event information, taken from a variable level, to the buffer.

```
#EMSINITV [ / option [ , option ] / ] buffer-var ssid
  event-number token-id source-var
```

option

is either of these:

SSID *ssid*

is the subsystem ID of the subsystem to which the subject token belongs. If omitted or zero (0.0.0), it defaults to the SSID of the subsystem generating the event.

TIMESTAMP *timestamp*

is the timestamp, in external representation of Julian GMT format, for the event message. If you omit this parameter, the current time is used.

buffer-var

is the name of a writable STRUCT to be used as an EMS buffer. #EMSINITV automatically passes the data length of the STRUCT to the EMSINIT procedure. Any current contents of the STRUCT are lost.

ssid

is the subsystem ID of the subsystem generating the event.

event-number

is a number, specific to this subsystem, that identifies this event message.

token-id

is the token code or token map of the subject of this event message.

source-var

is the name of a variable level, of type STRUCT, from which #EMSINITV is to obtain binary token values. The contents of the STRUCT are not altered.

Result

#EMSINITV returns a numeric status code indicating the outcome of the initialization. The meaning of the code:

Code	Description
0	No error
-2	Invalid parameter value
-3	Missing parameter
-4	Invalid parameter address
-5	Buffer full
-7	Internal error
-10	Invalid subsystem ID
-12	Insufficient stack space

Considerations

- If the data in *source-var* is longer than the data area expected by #EMSINITV, the excess bytes are ignored without any notification. If the data in *source-var* is shorter than the data area expected, #EMSINITV sets the remainder of the token value to unspecified values.
- To initialize a buffer and supply event information in external form, use the #EMSINIT built-in function.

#EMSTEXT Built-In Function

Use #EMSTEXT to obtain information from an event buffer and make it available in the function result as printable text derived from the event.

```
#EMSTEXT [ / option [ , option ] ... / ] buffer-var
```

option

is one of these:

INDENT *num-chars*

specifies the number of characters to indent the second and subsequent lines of text generated by the EMSTEXT procedure. INDENT 0 means no indentation. If you omit this option, or specify INDENT -1, EMSTEXT applies its own default indentation.

`INITTEMPLATE` *struct*

This option is reserved for use by the TACL software product.

`LINES` *num-lines*

specifies the number of lines in the print buffer. If you omit it, the default is 10 lines.

`OPRLOG` *num*

specifies the format of the printable information. If num is not zero, EMSTEXT searches for OPMSG and TEXT tokens in the event buffer and, if any are present, formats the printable result similarly to an operator log message. If none is found, EMSTEXT applies its own default formatting rules.

If you omit this option, or if num is zero, EMSTEXT uses its default formatting rules in preference to the OPRLOG format.

`WIDTH` *num-chars*

is the width of each line in the print buffer. If you omit this option, 80-character lines are assumed.

buffer-var

is the name of a STRUCT containing an SPI buffer.

Result

#EMSTEXT returns the formatted text from the event buffer.

Considerations

- You cannot access the lengths result from the EMSTEXT procedure with #EMSTEXT; you must use #EMSTEXTV instead.
- The INT(32) result of EMSTEXT is not included in the result of #EMSTEXT; you must use #EMSTEXTV to access it.
- Because #EMSTEXT can produce more than one line, you should always use square brackets around the construct containing the invocation of #EMSTEXT:

  ```
  [#OUTPUT [#EMSTEXT var]] and [#SET var2 [#EMSTEXT var1]]
  ```

  are correct, whereas

  ```
  #OUTPUT [#EMSTEXT var] and #SET var2 [#EMSTEXT var1]
  ```

  would produce TACL errors if #EMSTEXT returned more than a single line.
- #EMSTEXT does not suppress leading and trailing blank lines in the result, but remember that TACL ignores leading and trailing white space on arguments in most cases.
- #EMSTEXT does not suppress leading and trailing spaces on nonblank lines in the result, but remember that TACL ignores leading and trailing white space on arguments in most cases.
- If a STRUCT is longer than the data it is to contain, the unused bytes are ignored on input and unchanged on output.
- If a STRUCT is too short, an error results.
- To obtain information from an event buffer and place the printable text into a STRUCT, use the #EMSTEXTV built-in function.

#EMSTEXTV Built-In Function

Use #EMSTEXTV to obtain data from an event buffer and put the printable text derived from the event into a STRUCT.

```
#EMSTEXTV [ / option [ , option ] ... / ] buffer-var
  formatted-var [ lengths-var ]
```

option

is one of these:

```
INDENT num-chars
INITTEMPLATE struct
LINES num-lines
OPRLOG num
WIDTH num-chars
```

These have the same definitions as those in the #EMSTEXT built-in function.

buffer-var

is the name of a STRUCT containing an SPI buffer.

formatted-var

is the name of a STRUCT that is to receive the formatted text. The STRUCT can have any definition, but it must be large enough to contain at least *num-lines* of *num-chars* each.

lengths-var

is the name of a STRUCT that is to receive the lengths result from the EMSTEXT procedure; lengths indicates the actual length of each line in the printable text result. If, for example, the *formatted-var* STRUCT can contain 5 lines of 80 characters each, and the formatting of a given message places 75 characters in the first line, 28 characters in the second, and nothing at all in the remaining three lines, the EMSTEXT procedure puts [75, 28, -1, -1, -1] into *lengths-var*. This STRUCT can have any definition, but it must be at least large enough to contain 2 * num-lines characters.

Result

#EMSTEXTV returns a space-separated pair of signed integers representing the result of EMSTEXT. Their meaning is:

Result	Description
0 0	No error
0 22	Invalid parameter address or invalid parameter
0 29	Missing parameter

Result	Description
1 code	ALLOCATESEGMENT error code
2 +code	Template file open failed
2 -code	Template file opened, but has improper structure
3 code	Template file read error
4 0	*buffer-var* not a valid SPI buffer
5	code File read error (INITTEMPLATE related)
6 0	Malformed template
7 code	MOVEX error code

If error 0 22 or 0 29 occurs, no text appears in the STRUCT. In all other cases, #EMSTEXTV transfers text to *formatted-var*, although it might be incomplete or a default conversion.

Considerations

- If a STRUCT is longer than the data it is to contain, the unused bytes are ignored on input and unchanged on output.
- If a STRUCT is too short, an error results.
- To obtain information from an event buffer and obtain the printable text as the function result, use the #EMSTEXT built-in function.

#ENDTRANSACTION Built-In Function

Use #ENDTRANSACTION to commit the database changes associated with a transaction identifier. #ENDTRANSACTION invokes the ENDTRANSACTION operating system procedure. When this procedure is called by the process (or its backup) that issued #BEGINTRANSACTION, the TMF subsystem tries to commit the transaction. If it does so successfully, the changes made by the transaction are permanent, and the locks held for the transaction are released. Locks are held until #ENDTRANSACTION exits (or until #ABORTTRANSACTION occurs).

```
#ENDTRANSACTION
```

Result

#ENDTRANSACTION returns 0, if successful, or a file-system error indicating the reason ENDTRANSACTION failed.

Consideration

To abort and back out a transaction, use the #ABORTTRANSACTION built-in function.

#EOF Built-In Function

Use #EOF to set a flag so that a process receives one end-of-file after it reads all the data in a variable level. That variable level is one that is used either as the IN variable of a #SERVER or as the DYNAMIC IN variable (see the #NEWPROCESS Built-In Function on page 9-265) of an implicit server. If the variable level is not being used for either of these purposes, #EOF does nothing. After a process has read the end-of-file sent to it, the flag is cleared and subsequent read operations wait for more data or another #EOF.

```
#EOF variable-level
```

variable-level

is the name of a variable level used for DYNAMIC IN.

Result

#EOF returns nothing.

Consideration

Different processes handle end-of-file differently. For example, an EDIT process started by the command

```
EDIT /INV in_var DYNAMIC/
```

stops as soon as it reads an end-of-file produced by

```
#EOF in_var
```

#ERRORNUMBERS Built-In Variable

Use #ERRORNUMBERS to get information about the most recent error detected by your TACL.

```
#ERRORNUMBERS
```

Result

#ERRORNUMBERS returns a space-separated list of four integer numbers that describe the most recent error. If the first number in the list is less than 1024, it is the number of a file-system error or sequential I/O error. If the number is 1024 or larger, it is a TACL error number. The other three numbers are used in certain cases to provide additional information:

- If a #NEWPROCESS operation is unsuccessful, the first number is 1149. The second number indicates the specific error returned by the PROCESS_CREATE_ procedure. The third number contains error detail information from PROCESS_CREATE_. The fourth number is not used.
- If your TACL programs use #SET #ERRORNUMBERS in their exception handling, the numbers will be whatever you set them to.

Note. D-series TACL returns 1149 for a process creation failure; C-series and earlier RVUs of TACL return 1101.

Considerations

- When you first log on, #ERRORNUMBERS is initialized to 0 0 0 0.
- All syntax errors (“Expecting . . . ”) set #ERRORNUMBERS to 1048 0 0 0.
- Use #PUSH #ERRORNUMBERS (or PUSH #ERRORNUMBERS) to save a copy of all your current error numbers.
- Use #POP #ERRORNUMBERS (or POP #ERRORNUMBERS) to restore the four error numbers from the copy last pushed.
- Use #SET #ERRORNUMBERS (or SET VARIABLE #ERRORNUMBERS) to set the four TACL error numbers to the desired values.

The syntax for #SET #ERRORNUMBERS is:

```
#SET #ERRORNUMBERS n n n n
```

To create your own error categories, use any numbers in the space-separated list.

- For more information about error messages, see the *Guardian Procedure Calls Reference Manual* and the *Guardian Procedure Errors and Messages Manual*.

Example

If you try to create a process that already exists, you receive this error when you output the error numbers:

```
37> #OUTPUT [#ERRORNUMBERS]
1149 1 48 0
```

The first number is a TACL #NEWPROCESS error (1149). The second number is a PROCESS_CREATE_ error number (1), which usually indicates that a file system error occurred. The third number is a file-system error number (48), which indicates that a security violation occurred. The fourth number is not used.

#ERRORTEXT Built-In Function

Use #ERRORTEXT with exception handlers to intercept any error text that might have been written to OUT if there had been no exception handler.

```
#ERRORTEXT / option /
```

option

is one of these:

CAPTURE *variable-level*

indicates that the specified variable level is to be cleared and the current error text, if any, moved into it.

CLEAR

specifies that the current error text is to be discarded.

Result

#ERRORTEXT returns nothing.

Considerations

- If an error occurs while you are filtering _ERROR (see #EXCEPTION Built-In Function on page 9-163), TACL stores the resulting error text internally. Previously stored text is lost, so only the most recent error text is available.
- The captured error text is not fully formatted and usually contains some internal representations. It is, therefore, suitable only for output with #OUTPUTV or OUTVAR.

#EXCEPTION Built-In Function

Use #EXCEPTION, when exception handling is in effect, to determine why a routine was invoked.

```
#EXCEPTION
```

Result

#EXCEPTION returns _CALL, if the routine issuing #EXCEPTION was invoked normally, or returns the name of the exception if the routine was invoked in response to an exception that it had filtered. (See the *TACL Programming Guide* for a discussion of exception handling.)

#EXIT Built-In Variable

Use #EXIT to determine the current state of the exit flag.

```
#EXIT
```

Result

#EXIT returns 0 if the EXIT flag is off and -1 if it is on. If the EXIT flag is on, it causes TACL to exit as soon as its buffer becomes empty.

Considerations

- When you first log on, #EXIT is initialized to zero.
- Use #PUSH #EXIT (or PUSH #EXIT) to save a copy of your current setting of the exit flag.
- Use #POP #EXIT (or POP #EXIT) to restore the exit flag from the last copy pushed.
- Use #SET #EXIT (or SET VARIABLE #EXIT) to turn the exit flag on or off.

The syntax of #SET #EXIT is:

```
#SET #EXIT num
```

num

is zero to set the flag off, nonzero to set it on.

#EXTRACT Built-In Function

Use #EXTRACT to obtain the first line of a variable level.

```
#EXTRACT variable-level
```

variable-level

is the name or level of an existing variable level whose first line is to be extracted.

Result

#EXTRACT returns the first line of the specified variable level and deletes the line from the variable level.

Example

```
21> #PUSH fline
22> #SET fline [#EXTRACT linevar]
```

Consideration

To move the first line of a variable level to another variable level, use the #EXTRACTV built-in function.

#EXTRACTV Built-In Function

Use #EXTRACTV to move the first line of a variable level to another variable level. You can also use #EXTRACTV to set the binary data of a STRUCT directly, by extracting the first line from a variable level of another type and placing it in the STRUCT. No type checking is performed. (This use is primarily for accepting structured data from the variables associated with requesters and servers.)

```
#EXTRACTV from-variable-level to-variable-level
```

from-variable-level

is the name of an existing variable level; #EXTRACTV deletes the first line of this variable level.

to-variable-level

is the name of a variable level. #EXTRACTV puts the deleted line from *from-variable-level* into this variable level. If this variable already exists, its previous contents are lost. If this variable does not exist, an error occurs.

Result

#EXTRACTV returns nothing.

Considerations

- The construct

  ```
  #EXTRACTV var struct
  ```

  is not equivalent to

  ```
  #SET struct [#EXTRACT var]
  ```

The former construct is faster and is not subject to errors resulting from type checking.

- If the data supplied exceeds the data area of the specified STRUCT, TACL discards the excess bytes without any notification.
- If the data supplied is shorter than the data area of the STRUCT, TACL sets the entire STRUCT to default values and then moves the data into the beginning of the data bytes of the STRUCT.
- To retrieve the first line of a variable level as the function result, use the #EXTRACT built-in function.

#FILEGETLOCKINFO Built-In Function

Use #FILEGETLOCKINFO to obtain information about record and file locks.

```
#FILEGETLOCKINFO [ / option / ] fvname control lockdesc
  participants
```

option

can be one of the following options or can be omitted. If omitted, #FILEGETLOCKINFO returns information for all locks.

PROCESSID *processid*

specifies the name or PID (if not named) of an existing process. The #FILEGETLOCKINFO built-in function will return information about locks held or requested by the specified process.

TRANSID *transid*

returns information about locks held or requested by the specified TMF subsystem transaction. *transid* is a four-word transaction identifier that uniquely identifies a specific transaction protected by the TMF subsystem. *transid* must have this format:

\node (*crash-count*). *cpu. sequence-number*

(*crash-count*) can be omitted if it equals 0. *node* can specify a node name or node number and can be omitted for a local node if *crash-count* is also omitted. For additional information about transaction identifiers, see the TMF subsystem documentation.

fvname

specifies the name of an existing file or volume for which lock information is desired. If the PROCESSID or TRANSID is specified, *fvname* must be a volume name; otherwise, it can be a volume name or a file name. If the name is not fully qualified, the current defaults apply.

control

is a STRUCT used to control a series of calls to #FILEGETLOCKINFO. This STRUCT is updated by #FILEGETLOCKINFO on each call. The contents are for use by the operating system and should not be modified by the user.

The STRUCT must be defined as an array of ten integers. Example:

```
[#DEF control STRUCT
  BEGIN
    INT x (0:9);
  END;
]
```

The control STRUCT must be initialized prior to a series of calls to #FILEGETLOCKINFO. This statement initializes the sample STRUCT defined previously:

```
#SET control 0
```

lockdesc

is a STRUCT that is to receive the lock information; it must be defined as follows:

```
[#DEF lockdesc STRUCT
  BEGIN
    INT lock^type;      == 0 = file, 1 = record
    UINT flags;         == <0> set if generic lock
                        == <1:15> are reserved
    INT n^participants; == number of holders/waiters
                        == for lock
    INT2 record^id;     == if record lock and not
                        == key-sequenced
    INT key^length;     == for key-sequenced record
                        == locks; 0 if not
                        == key-sequenced
    CHAR key(0:255);    == key for key-sequenced
  END;
]                     == record locks
```

participants

is a STRUCT that is to receive information about processes or transactions that hold or wait for the lock; it must be defined as follows:

```
[#DEF participants STRUCT
  BEGIN
  STRUCT locker(0:mp-1); == mp = max participants
    BEGIN
      UINT flags;        == <0> set for process
                         == clear for
                         == transaction
                         == <1:3> 0=waiting,
                         == 1=granted
                         == <4> internal use
                         == <5:15> reserved
      FILLER 2;          == reserved
      PHANDLE process;   == process holding or
                         == waiting for the lock
      TRANSID transid    == transaction holding or
      REDEFINES process; == waiting for the lock
    END;
  END;
]
```

Result

The #FILEGETLOCKINFO built-in function returns a number that indicates the status of the operation.

Status codes are listed in Table 9-10.

Table 9-10. #FILEGETLOCKINFO Status Codes

Code	Meaning
0	Success
-1	The control STRUCT is of the wrong size.
-2	The lockdesc STRUCT is of the wrong size.
-3	The participants STRUCT is of the wrong size. (This STRUCT can vary in size, but it must always be a multiple of the size that would be required for one participant.)
45	The participants STRUCT was too small to hold all the lock holders and waiters for the locked resource in the lock description STRUCT. The call was successful, but only the number of participants specified by max participants were returned.
1	No more locks are available. No information was returned.
0	Success
-1	The control STRUCT is the wrong size.
-2	The lockdesc STRUCT is the wrong size.
-3	The participants STRUCT is the wrong size. (This STRUCT can vary in size, but it must always be a multiple of the size that would be required for one participant.)

If #FILEGETLOCKINFO is called to find locks on a disk volume and the status code is 0 (no error) or 45 (warning; there were more lock holders or waiters than would fit into the participants structure), the status number is followed by a space and the name of the locked file. The file name is in the form subvolume.file, without a node name or volume name.

Considerations

- #FILEGETLOCKINFO replaces #LOCKINFO for D-series operating systems. Existing #LOCKINFO calls should be converted to #FILEGETLOCKINFO calls for D-series use.
- A file or record can have one granted lock and several pending locks. Each call to #FILEGETLOCKINFO returns information about one lock and as many holders or waiters as permitted by the size of your participants STRUCT. To obtain information on all locks on a file or volume, make successive calls to #FILEGETLOCKINFO until the returned status code equals 1. On each call, the file system updates the control STRUCT and keeps track of information for the next call.
- #FILEGETLOCKINFO uses the D-series FILE_GETLOCKINFO_ procedure.

#FILEINFO Built-In Function

Use #FILEINFO to obtain detailed, specific information about a file.

```
#FILEINFO / option [ , option ] ... / file-name
  [ file-name ... ]
```

option

is an information request. It can be any of these:

AUDITED

returns -1 if the file exists and is audited by the TMF subsystem; otherwise, it returns 0.

BLOCKLENGTH

returns one of these:

- For structured files, the physical block length (in bytes)
- For unstructured files, the length of the system buffer used for the file (in bytes)
- For non-existent files, 0

CODE

returns the numeric file code (0, if the file does not exist); see the FUP INFO command in the *File Utility Program (FUP) Reference Manual*. This output includes file type 800, the native object file for TNS/E systems.

CORRUPT

returns -1 if the file exists and has been corrupted; otherwise, it returns 0. (If a FUP DUP operation is writing to the file, it is considered to be corrupt until the operation is finished.)

CRASHOPEN

returns -1 if the file is marked crash-open, meaning the file was not closed normally by the disk process; otherwise, it returns 0.

CREATION_GMT

returns the time and date when the file was created, expressed as a four-word timestamp (0, if the file does not exist or if it was created prior to the B40 RVU).

DATACOMPRESSION

returns -1 if the file exists, is key-sequenced, and uses data compression; otherwise, it returns 0.

DEVICETYPE

returns the numeric device type word (0, if the file does not exist); see the *Guardian Procedure Calls Reference Manual*.

EOF

returns the numeric value of the end-of-file pointer, which represents the size of the file in bytes (0, if the file does not exist).

EXISTENCE

returns -1 if the file exists; if not, it returns 0.

EXTENTSALLOCATED

returns the number of file extents currently allocated (0, if the file does not exist).

FILE

returns the file-name portion of the fully expanded file name (missing fields in file-name are supplied by the current defaults); the file need not exist. If the file is a temporary file, FILE returns nothing.

FILEFORMAT

returns one of these:

1 if the file is a format 1 file

2 if the file is a format 2 file

0 if the file does not exist

FILESTRUCTURE

returns a numeric value that indicates the structure of the file:

0 Unstructured (or nonexistent)

1 Relative

2 Entry-sequenced

3 Key-sequenced

FULLNAME

returns the fully expanded file name (missing fields in file-name are supplied by the current defaults). The define name is returned if a define name is specified. The file need not exist.

INCOMPLETESQLDDL

returns -1 if the file exists and an SQL DDL operation is currently in progress. Otherwise, it returns 0.

INDEXCOMPRESSION

returns -1 if the file exists, is key-sequenced, and uses index compression; otherwise, it returns 0.

LASTOPEN_GMT

returns the time and date when the file was last opened, expressed as a four-word timestamp. (If the file does not exist, or has not been opened since before the B20 software RVU, this option returns 0.)

LICENSED

returns -1 if the file exists and is licensed; otherwise, it returns 0.

MAXBYTES

returns the maximum number of bytes configured for the file (0, if the file does not exist).

MAXEXTENTS

returns the maximum number of file extents configured for the file (0, if the file does not exist).

MODIFICATION

returns the numeric modification date (a three-word timestamp); if the file does not exist, this option returns 0.

When a file is copied from one system to another system in a different time zone without modifying the source date of the file, the MODIFICATION option returns the file modification time as the local civil time of the system from which the file was copied.

MODIFICATION_LCT

returns the time and date when the file was last modified, expressed as a four-word timestamp. The time is expressed in the local civil time (LCT) of the system on which the file resides. If the file does not exist, returns 0.

When a file is copied from one system to another system in a different time zone without modifying the source date of the file, the MODIFICATION_LCT option returns a timestamp that shows the file modification time as the local civil time of the system to which the file was copied.

ODDUNSTR

returns -1 if the file exists, is unstructured, and was created with the ODDUNSTR parameter in effect; otherwise, it returns 0.

OPENNOW

returns -1 if the file exists and is currently open; otherwise, it returns 0.

OWNER

returns the owner's user ID (0,0 if the file does not exist).

PHYSICALFILENAME

returns the physical file name that is associated with the physical file where the data resides if the specified file is a logical file. If the file is a remote file, the remote system name is also returned. If the file does not exist, nothing is returned.

PHYSICALVOLUME

returns the physical volume name that is associated with the physical file where the data resides if the specified file is a logical file. If the file is a remote file, the remote system name is also returned. If no physical volume name has been specified, the physical volume is chosen by the system and this name is returned. If the file does not exist, nothing is returned.

PRIMARY

returns the primary extent size (0, if the file does not exist).

PROGID

returns -1 if the file exists and has PROGID authority; otherwise, it returns 0.

RECORDLENGTH

returns the logical record length for the file (0, if the file does not exist or is unstructured).

REFRESH

returns -1 if the file exists and its refresh flag is set; otherwise, it returns 0.

SECONDARY

returns the secondary extent size (0, if the file does not exist).

SECURITY

returns the security established for the file:

rwep	for a Guardian file
****	for a Safeguard file
####	for an OSS file
*SQL	for SQL/MX objects
blank	for a nonexistent file

SUBVOL

returns the subvolume-name portion of the file name (missing fields in file-name are supplied by the current defaults); the file need not exist. If the file is a temporary file, SUBVOL returns the file-name portion (#*nnnn*).

SYSTEM

returns the node-name portion of the file name (missing fields in file-name are supplied by the current defaults); the file need not exist. If neither the defaults nor file-name includes a node name, SYSTEM returns nothing.

UNRECLAIMEDFREESPACE

returns -1 if the file exists and an SQL DDL operation has left unreclaimed free space. Otherwise, it returns 0.

VOLUME

returns the volume-name portion of the file name, using the current defaults if not specified in the file name. The define name is returned if a define name is specified. The file need not exist.

file-name

is the name of the file. More than one file can be specified. When specifying a file name, TACL requires a subvolume name if you supply a volume name. To obtain the volume portion of the current defaults, use this command:

```
#FILEINFO /VOLUME/ [#DEFAULTS].X
```

The file X does not need to exist.

Result

#FILEINFO returns the information requested.

Considerations

- If you specify more than one request option, the items of information are listed, separated by spaces, in the same order as the requests.
- Because FILE and SYSTEM can return nothing, you should specify them last in the list of options to avoid confusion. When assigning results with #SETMANY, for example, you avoid loss of synchronization between the items returned and the variable levels that receive them.
- If the RVU of the system (system procedure FILE_GETINFOLISTBYNAME_) does not support INCOMPLETESQLDDL, UNRECLAIMEDFREESPACE, PHYSICALVOLUME, or PHYSICALFILENAME, this message is generated:

  ```
  *ERROR* FILE_GETINFOLISTBYNAME_ error = 561
  ```

#FILENAMES Built-In Function

Use #FILENAMES to obtain all file names that satisfy a specified file-name template.

```
#FILENAMES [ / option [ , option ] ... / ]
  [ file-name-template ]
```

option

specifies the criteria the requested file names must meet; specify one of these options:

`MAXIMUM [ num ]`

specifies the maximum number of file names to be returned on this call. If num is omitted or is less than 1, all matching file names are returned. If num is greater than 0, no more than num file names are to be returned; all file names returned are in the same subvolume.

`NEWSUBVOL variable-level`

specifies the name of an existing variable level that is to indicate whether the file-name search started in a new subvolume. If you include PREVIOUS, and if the first file name returned is in the same subvolume as the file name specified by PREVIOUS, the variable level is set to 0. In all other circumstances, the variable level is set to -1.

`PREVIOUS [ file-name ]`

specifies a starting point, alphabetically, after which file names are to be matched. It need not specify an existing file. If you omit *file-name*, file-name matching starts with the first file that matches file-name-template.

file-name-template

is a file name. You may include these template characters in any field of the file specification except the system field:

* matches zero or more characters.

? matches a single character.

Template characters cannot match a volume identifier ($) or a field separator (.).

If you omit *file-name-template*, #FILENAMES applies to file names in your current subvolume.

Result

#FILENAMES returns a space-separated list of file names.

Consideration

File names within a volume are returned in alphabetical order, but volumes are processed in logical-device number order.

Examples

This example illustrates the output of the #FILENAMES built-in function:

```
17> #OUTPUT [#FILENAMES $BOOKS.TACL.SEC*]
$BOOKS.TACL.SEC01 $BOOKS.TACL.SEC02 $BOOKS.TACL.SEC03
18>
```

The asterisk in the file-name template instructs #FILENAMES to display the names of all files that begin with SEC, regardless of which, or how many, characters follow. This example illustrates the use of two file-name template characters. The asterisk (*) in the subvolume indicates that you want TACL to search for any subvolume that begins with BOOK. The question mark (?) in the file name indicates that you want to list the name of any file whose name has SECT in the first four positions, any character in the fifth position, and a 2 in the sixth position:

```
18> #OUTPUT [#FILENAMES book*.sect?2]
BOOK1.SECT02 BOOK2.SECT12
19>
```

#FILTER Built-In Function

Use #FILTER in a routine to specify the exceptions the routine can handle. An exception is an unusual event that causes TACL to interrupt the normal flow of invocations and transfer to special code, known as an exception handler. The event can be BREAK, a TACL error, or a user-defined exception.

```
#FILTER [ exception [ exception ] ... ]
```

exception

is the name of an unusual event, which can be any of these:

_BREAK

specifies that the BREAK key is to be handled as an exception, when pressed.

_ERROR

specifies that errors detected by TACL are to be handled. In addition, your code can explicitly raise this exception.

user-exception

specifies a user-defined routine that can be invoked by #RAISE, causing the routine that handles #FILTER to be reinvoked and to set #EXCEPTION to user-exception.

Result

#FILTER returns nothing.

Considerations

- A portion of a TACL program that performs the actions required by one or more exceptions is known as an exception handler. An exception handler must do the following, within a routine:
  - Include a #CASE built-in function call to handle each of the exceptions the handler supports. The #EXCEPTION built-in function returns the type of exception that invoked the handler; use it to direct the #CASE function.
  - Specify names of exceptions supported by the handler. For this step, use the #FILTER built-in function to specify the exceptions. TACL does not invoke an exception handler for an exception unless it is listed in the #FILTER list.

If your code contains these two items and an exception occurs while one of your functions is executing, the TACL process will invoke the #EXCEPTION built-in function with the name of the exception that occurred.

- Only the most recent #FILTER in a routine has any effect. If a routine that contains a filter list calls another routine that also contains a filter list, the list in the called routine becomes active. When control is returned from that routine, the filter list in the calling routine resumes active status. If a routine contains more than one #FILTER function call, the latest list supersedes all previous lists.
- When an exception occurs, if the current routine has no exception handler, TACL automatically exits that routine and pops you out of routines until it finds a #FILTER for that exception and reinvokes the routine containing the #FILTER. If TACL finds no such routine, it performs its normal exception handling: it resets frames and results and, if the exception is the predefined _ERROR exception, it displays an error message.
- To determine why an exception handler was invoked, use the #EXCEPTION built-in function. To raise an exception, use the #RAISE built-in function.

Example

This example shows a #FILTER call that lists two exceptions:

```
#FILTER _ERROR userex
```

For examples showing the use of #FILTER with exception handlers, refer to the *TACL Programming Guide*.

#FRAME Built-In Function

Use #FRAME to establish a reference point for pushed variables. A subsequent occurrence of #UNFRAME pops all variable levels pushed since the last #FRAME.

```
#FRAME
```

Result

#FRAME returns nothing.

Considerations

- All variables pushed after a #FRAME are automatically popped when a matching #UNFRAME is executed. This allows you to control local variables without having to explicitly pop them.
- Nesting of frames is allowed.
- Use #VARIABLEINFO with the / FRAME / option to determine the frame level of a particular variable.
- #FRAME does not restrict the variable levels you can access. You can always refer to all existing variable levels regardless of your frame.
- If an error occurs in a macro or routine, TACL restores the frame level of all variables to the level in effect when the macro or routine was invoked.

#GETCONFIGURATION Built-In Function

Use #GETCONFIGURATION to obtain the settings of special TACL flags that can change the behavior of TACL in certain situations.

Note. These flags can be changed only by users who are authorized to alter the Command Interpreter Monitor (CMON). For additional information, see Command Interpreter Monitor Interface (CMON) on page 6-8.

```
#GETCONFIGURATION / option [ , option ] ... /
```

option

can be any of these:

AUTOLOGOFFDELAY

if greater than zero, specifies the maximum number of minutes that TACL is to wait at a prompt. If that time is exceeded, TACL automatically logs off, releases the default segment, and (if LOGOFFSCREENCLEAR so specifies) clears the terminal screen. This option is initially set to -1 (disabled).

The TACL process issues a modem disconnect at autologoff.

BLINDLOGON

if not zero, specifies that the LOGON command, whether in the logged-off state or the logged-on state, prohibits the use of the comma, requiring the password to be entered at its own prompt while echoing is disabled. The setting does not change the behavior of #CHANGEUSER. This option is initially set to 0.

CMONREQUIRED

if not zero, specifies that all operations requiring approval by $CMON are denied if $CMON is not available or is running too slowly. Approval of $CMON is not required if the TACL is already logged on as the super ID. This option is initially set to 0.

Caution. If you are authorized to change CMONREQUIRED and intend to set it to a nonzero value, you must keep an unmodified copy of TACL for system operation use; otherwise, you cannot log on if $CMON is not running or is running too slowly. If the modified TACL is in $SYSTEM.SYSnn, you cannot even start the system.

CMONTIMEOUT

specifies the number of seconds that TACL is to wait for any $CMON operation. This option is initially set to 30.

CONFIGRUN

if PROCESSLAUNCH, then the TACL RUN[D] command was configured to call the system procedure PROCESS_LAUNCH_ to start the process. In this case, the additional RUN[D] command options, MAXMAINSTACKSIZE, MAXNATIVEHEAPSIZE, and GUARANTEEDSWAPSPACE, are available.

if PROCESSCREATE, then the TACL RUN[D] command was configured to call the system procedure PROCESS_CREATE_ to start the process. In this case, thee additional RUN[D] command options, MAXMAINSTACKSIZE, MAXNATIVEHEAPSIZE, and GUARANTEEDSWAPSPACE, are not available.

LOGOFFSCREENCLEAR

if not zero, specifies that if TACL is interactive, and the IN file is a 65xx terminal, the terminal memory is cleared at (#)LOGOFF unless the NOCLEAR option is supplied. The CLEAR and NOCLEAR options always override the automatic operation. This option is initially set to -1.

NAMELOGON

if not zero, specifies that the LOGON command, in both the logged-off and logged-on states, and #CHANGEUSER do not accept user numbers but require that user names be used. This option is initially set to 0.

NOCHANGEUSER

if not zero, TACL disables the ability to log on from a logged-on state. This option is initially set to 0.

REMOTECMONREQUIRED

if not zero, specifies that all operations requiring approval by a remote $CMON are denied if that remote $CMON is unavailable or is running too slowly. $CMON approval is not needed if the TACL is already logged on as the super ID. This option is initially set to 0.

REMOTECMONTIMEOUT

specifies the number of seconds that TACL is to wait for any $CMON operation involving a remote $CMON. This option is initially set to 30.

REMOTESUPERID

if zero, specifies that if TACL is started remotely, any attempt to log on (or use #CHANGEUSER) with the super ID (255,255) results in an illegal logon. This option is initially set to -1 (disabled).

REQUESTCMONUSERCONFIG

if not zero, after a LOGON command or #CHANGEUSER built-in function is executed, the TACL process sends a message to the $CMON process

requesting the configuration parameters (the flag settings) in effect for the current user.

This option is meaningful only if the $CMON process has been coded to return the appropriate information. Otherwise the request is ignored. Use of this option allows $CMON to control (and change) the flag settings for each logon session. These flag settings are then returned to the TACL process.

This option is initially set to 0.

Regardless of the setting, when a LOGOFF command has been executed and followed (at some point) by a LOGON command or when a noninteractive TACL process is started, the TACL process requests the configuration information from the $CMON process.

`STOPONFEMODEMERR`

if not zero, specifies that the TACL process stops when error 140 (FEMODEMERR) is encountered on its input. If the TACL process is started with the PORTTACL startup parameter, then this TACL configuration setting is ignored (TACL goes to the logged off state and waits for a modem connect when error 140 is encountered).

The option is initially set to 0, meaning the TACL process is put in a logged off state and waits for a modem connection when an error 140 is encountered.

Result

#GETCONFIGURATION returns a space-separated list of the selected values in the order of their request.

#GETPROCESSSTATE Built-In Function

Use #GETPROCESSSTATE to obtain process state information for the current TACL process.

```
#GETPROCESSSTATE [ / option [ , option ] ... / ]
```

option

can be zero or more of these:

LOGGEDON

requests the logged-on state of the TACL process. A value of 1 indicates that the TACL process is in a logged-on state and that the TACL process default file security, process access ID (PAID), creator accessor ID (CAID), and remote logon flags were set to their appropriate values by the operating system.

The LOGGEDON option always returns a value of 1, because a TACL process must be in the logged-on state to invoke the #GETPROCESSSTATE built-in function.

TSNLOGON

requests information about whether or not Safeguard software authenticated the TACL process. A value of 1 indicates that Safeguard software authenticated and created the TACL process or that the TACL process is a descendent of a local process that had the TSNLOGON flag set. A value of 0 indicates that Safeguard software did not authenticate or create the TACL process.

The default value is 0 for a regular TACL process and 1 for a TSN-TACL process or a TACL process started by a TSN-TACL process.

TSNLOGOFF

requests information about authentication and logged-on state of the TACL process. A value of 1 indicates that Safeguard software authenticated the TACL process and that the process was subsequently logged off.

The TSNLOGOFF option always returns a value of 0, because a TACL process must be in the logged-on state to invoke the #GETPROCESSSTATE built-in function.

INHERITEDLOGON

requests information about authentication of the TACL process. A value of 1 indicates that the TACL creator authenticated the user of the TACL process. When this occurs, the TACL process skips user authentication and the process starts in the logged-on state. A value of 0 indicates that the process that created the TACL process did not authenticate the user prior to creating the

TACL process, so the TACL process performs user authentication after prompting for logon information.

The default value is 0 for a regular TACL process or a TACL process started by a TSN-TACL process. The default value is 1 for a TSN-TACL process.

STOPONLOGOFF

requests information about whether the current TACL process will be stopped after it enters a logged-off state. A value of 1 indicates that the current TACL process will be stopped when it enters a logged-off state. A value of 0 indicates that the current TACL process will not be stopped; instead, the TACL process will prompt for another logon.

The default value is 0 for a regular TACL process and 1 for a TSN-TACL process or a TACL process started by a TSN-TACL process.

PROPAGATELOGON

requests information about how local child TACL processes start. A value of 1 for the current TACL process indicates that the INHERITEDLOGON state of the child TACL process will be 1. The new process will start in the logged-on state. A value of 0 for the current TACL process indicates that the INHERITEDLOGON state of the local child TACL process will be 0. The new process will start in the logged-off state.

The default value is 0.

For remote child TACL processes, the parent process propagates a value of 0 for the INHERITEDLOGON flag.

PROPAGATESTOPONLOGOFF

returns information about how local child TACL processes stop. A value of 1 for the current TACL process indicates that the STOPONLOGOFF state of the child TACL process will be 1. The child process will stop when it enters a logged-off state. A value of 0 for the current TACL process indicates that the STOPONLOGOFF state of a local child process will be 0. The child process will prompt for logon information when it enters a logged-off state.

The default value is 0.

For remote child TACL processes, the parent process propagates a value of 0 for the STOPONLOGOFF flag.

Results

The #GETPROCESSSTATE built-in function returns a space-separated list of the selected values, in the order of their request.

If you do not specify any options, #GETPROCESSSTATE returns a space-separated list with a one or zero for each process state option, specified from left to right as

LOGGEDON, TSNLOGON, TSNLOGOFF, INHERITEDLOGON, STOPONLOGOFF, PROPAGATELOGON, and PROPAGATESTOPONLOGOFF.

If the #GETPROCESSSTATE built-in function cannot obtain process state information, the TACL process displays an operating system error and abends.

Considerations

- The term TSN-TACL refers to a TACL process that Safeguard software starts after authenticating the user of the TACL process.
- The TSNLOGON flag is set as follows for descendent TACL processes:
  - When a descendent TACL process is created on the same system as the parent process, the value of the TSNLOGON flag is propagated from the parent process to the descendent TACL process.
  - When a descendent TACL process is created on a remote system, the TSNLOGON flag is set to 0 (by the operating system) for that remote process.
- The state of PROPAGATESTOPONLOGOFF in the local parent process determines the initial value of STOPONLOGOFF in the current TACL. If the parent TACL process is remote, a value of 0 is propagated to the child process.
- To stop TACL when it enters a logged-off state, use the #SETPROCESSSTATE built-in function to set STOPONLOGOFF.
- In general, use the #SETPROCESSSTATE built-in function to alter process states as allowed by security guidelines.
- For more information about process state date, see the *Guardian Procedure Calls Reference Manual*.

#GETSCAN Built-In Function

Use #GETSCAN to obtain the number of characters that #ARGUMENT has processed.

```
#GETSCAN
```

Result

#GETSCAN returns the number of characters that #ARGUMENT has passed over, not including the routine name and the first character after the name.

Consideration

To determine whether there are more arguments in an argument set, use the #MORE built-in function. To specify a starting position at which the next #ARGUMENT function is to resume processing arguments, use the #SETSCAN built-in function.

#HELPKEY Built-In Variable

Use #HELPKEY to retrieve the name of the TACL help key. The help key displays syntax information during an interactive TACL session. You can type part of a statement, including the full name of a command or built-in function, and then press the help key. TACL lists what types of elements are expected next.

```
#HELPKEY
```

Result

#HELPKEY returns the name of the current help key or, if there is no help key, returns nothing.

Considerations

- When you first log on, #HELPKEY is initialized to F16.
- Defining a function key as the help key prevents that key from being programmed for any other function.
- Use #PUSH #HELPKEY (or PUSH #HELPKEY) to save a copy of the name of the current help key.
- Use #POP #HELPKEY (or POP #HELPKEY) to restore the help key from the last copy pushed.
- Use #SET #HELPKEY (or SET VARIABLE #HELPKEY) to define a function key as the help key.

The syntax of #SET #HELPKEY is:

```
#SET #HELPKEY [ key-name ]
```

key-name

is the name of the function key (such as F1 or SF13). If you omit *key-name*, you have no help key.

#HIGHPIN Built-In Variable

Use the #HIGHPIN built-in variable to establish the default PIN range for processes started by the current TACL.

```
#HIGHPIN
```

Considerations

- The default value for #HIGHPIN depends on the forced-low attribute of the TACL process. To ensure a default value for #HIGHPIN, set it to the desired value in the TACLLOCL or TACLCSTM file.
- The #HIGHPIN built-in variable can be pushed, popped, and expanded like any other built-in variable.
- The HIGHPIN option on a RUN command overrides the value of the #HIGHPIN variable for the new process. TACL uses the #HIGHPIN setting when there is no HIGHPIN directive on a RUN command or #NEWPROCESS call.
- Use #PUSH #HIGHPIN (or PUSH #HIGHPIN) to save the current #HIGHPIN setting.
- Use #POP #HIGHPIN (or POP #HIGHPIN) to restore #HIGHPIN to its previous identity, the last copy pushed.
- Use #SET #HIGHPIN (or SET VARIABLE #HIGHPIN) to specify the default PIN range.

The syntax of #SET #HIGHPIN is:

```
#SET #HIGHPIN { ON | OFF }
```

`ON`

specifies that a process will run at a high PIN if these conditions exist:

- The HIGHPIN bit is enabled in the object file.
- The HIGHPIN bit is enabled in the library file, if any.
- A high PIN is available.

`OFF`

specifies that processes run at a low PIN regardless of any other considerations.

#HISTORY Built-In Function

Use #HISTORY to display and manipulate the commands saved in the history buffer.

```
#HISTORY [ / option [ , option ] ... / ]
```

option

is an option that specifies the action to perform on the commands in the history buffer; it can be any of these:

CLEARLAST

deletes the most recent command line from the history buffer. For example, the TACL LOGON routine uses #HISTORY /CLEARLAST/ to erase the last line from the history buffer if the password is entered on the same line as the user ID. This action keeps the password confidential.

When entered from the IN file of a TACL process, #HISTORY /CLEARLAST/ removes itself from the history buffer. If included in a TACL program, #HISTORY /CLEARLAST/ removes the line entered from the IN file that caused the #HISTORY /CLEARLAST/ function to be executed.

RESET

deletes all command lines from the history buffer. The history number for the next TACL prompt is set to 1.

SHOW *num*

displays the last num command lines in the history buffer.

Result

#HISTORY with the CLEARLAST or RESET options returns nothing. #HISTORY with the SHOW option returns the last num commands. #HISTORY with no options returns all previous command lines that are still in the history buffer.

Considerations

- The history buffer is 1000 characters long; it contains zero or more lines. Each line stored in the history buffer requires as many bytes as the line contains, plus one extra byte.
- #HISTORY within a TACL macro or routine shows the lines in the history buffer that lead to the invocation. The result does not include the lines in the macro itself.

#HOME Built-In Variable

Use #HOME to set or obtain the variable defined as your home directory.

```
#HOME
```

Result

#HOME returns the full path-name of the home directory.

Considerations

- When you first log on, #HOME is initialized to : (the root directory).
- Use #PUSH #HOME (or PUSH #HOME) to save a copy of the name of your home directory.
- Use #POP #HOME (or POP #HOME) to restore your home directory to its previous identity, the last copy pushed.
- Use #SET #HOME (or SET VARIABLE #HOME) to define a directory variable as your home directory.

The syntax of #SET #HOME is:

```
#SET #HOME directory
```

directory

is the name of an existing variable level of type DIRECTORY.

#IF Built-In Function

Use #IF to obtain one set of text (typically TACL statements), contained within the #IF construct, if a given condition is met.

```
#IF [ NOT ] numeric-expression [ enclosure ]
```

NOT

specifies that #IF should negate *numeric-expression* after evaluating it; that is, the condition is met if *numeric-expression* is NOT true.

numeric-expression

is a decimal number, the name of a variable level containing a decimal number, or an arithmetic, relational, or logical expression that evaluates to true or false:

`true` = a numeric value other than zero

`false` = zero

enclosure

is an enclosure containing a THEN or an ELSE label. Each label is optionally followed by text that typically consists of one or more functions that can be executed if the option is chosen.

Result

If the expression evaluation is true, #IF returns the part of the enclosure labeled |THEN|, if it exists. If the expression is evaluated as being false, #IF returns the part of the enclosure labeled |ELSE|, if it exists. Any label not chosen is ignored.

Considerations

- #IF statements can be nested.
- The |THEN| and |ELSE| must be outside all square brackets within the enclosure or they are not recognized.
- You can omit both |THEN| and |ELSE| labels to make, in effect, a “don’t care” condition. For example, the #IF can be used in this way as a carrier for another function, deleting any numeric result of that function by testing it without acting on it.
- As with any function that contains an enclosure, the entire #IF construct must be enclosed in square brackets.

Examples

1. The macro, furd, outputs each of its arguments on a separate line:

```
[#DEF furd MACRO |BODY|
  == Output current argument
  #OUTPUT %1%
  == Test for more arguments
  [#IF NOT [#EMPTY %2 TO *%] |THEN|
    furd %2 TO *% == Call self without 1st arg
  ]
]
```

2. This example illustrates the use of a numeric expression:

```
[#IF 3 | THEN | . . .
```

3. This example illustrates the use of a variable name containing a decimal number:

```
#SET var 3
[#IF var | THEN | . . .
```

4. This example illustrates the use of an arithmetic expression enclosed in parentheses:

```
#SET var 4
[#IF (var-1 = 3) | THEN | . . .
```

5. This example shows an #IF with a contingent control path, to be taken if the numeric expression is false:

```
#SET tversion [#TACLVERSION]
#CHARGETV tversion tgeneric 1 TO 6
[#IF tgeneric '<' "T9205C" |THEN|
  SINK [#LOAD /KEEP 1/ $vol.subvol.firstlib]
  .
  .
  .
|ELSE|
  ATTACHSEG SHARED mysegf :mydir
  USE :mydir
]
```

6. This example illustrates an #IF with no enclosure, deleting the result of #ARGUMENT while still allowing #ARGUMENT to accomplish its action, assigning the argument value to NUM:

```
#IF [#ARGUMENT /VALUE num/ NUMBER]
```

#IN Built-In Variable

Use #IN to specify or obtain the name of the file currently being used by TACL as its IN file.

```
#IN
```

Result

#IN returns the name of the file currently being used for input by TACL.

Considerations

- At TACL startup time, #IN contains the name of the file currently being used as the TACL IN file. This file is called the primary IN file. To temporarily receive commands from an alternate source, push #IN and set #IN to another file or device. To restore the primary IN file, pop #IN.
- The contents of the #IN built-in variable determine the file or process from which TACL reads commands.
- TACL can store two IN file settings:
  - At startup time, the #IN built-in variable contains the name of the file currently in use as the TACL IN file. For an interactive terminal, #IN is initialized to the name of your home terminal. This is considered the primary IN file. You cannot permanently change the IN file for TACL; that is, the primary IN file always remains the same.
  - When you set #IN to a file other than the primary file, this file is known as the current IN file.
- The default IN file for processes run by TACL is not affected by #IN; it remains associated with the original TACL IN file. You can, however, use #INPUT or #INPUTV with the current IN file established by #SET #IN.
- When you use #SET #IN, it does not take effect until the next time TACL prompts for a command. Therefore, if you include a #SET #IN in a macro or routine, lines are not read from the new IN file until all lines resulting from the macro or routine have been processed.
- To set #IN to a disk file, TACL must allocate one of its block buffers internally. Because these block buffers are large and must be allocated from the first 64,000 bytes in the TACL address space, there are only four of them. (Other consumers of these blocks include the #OUT and #REQUESTER built-in functions.)
- Any error, including EOF, on a pushed #IN variable causes #IN to be popped at once. Any other error or break occurring while #IN is pushed causes all but the bottom level to be popped from #IN.

- Be careful to coordinate functions that enable the communication (such as #IN or #OUT) with the counterpart mechanisms in that process (such as IN or OUT
- referring to the TACL process name). Combining different mechanisms can lead to conflicts in interprocess message handling. For more information, see the *TACL Programming Guide*.
- When TACL is started with the IN file set to $RECEIVE, these guidelines apply:
  - You can set #IN without first pushing it. This feature, in conjunction with an equivalent one in #OUT, allows you to run TACL as a server, yet be able to take over a terminal as though TACL had been run on the terminal in the usual way.
  - When you #SET #IN without pushing it, TACL responds to the BREAK key if the device to which you set #IN supports BREAK.
  - The value of #IN is the default IN file for all processes started while #IN is set in this way.
  - You can revert to reading commands from $RECEIVE by setting #IN to nothing.
- Use #PUSH #IN (or PUSH #IN) to save the name of the currently open TACL IN file.
- Use #POP #IN (or POP #IN) to restore the last IN file name pushed.
- Use #SET #IN (or SET VARIABLE #IN) to name the file to be used by TACL as its IN file.

The syntax of #SET #IN is:

```
#SET #IN file-name
```

file-name

is the name of the IN file to be set. You can use the name of a process in place of a disk file name; TACL reads input from the process as if it were a file.

Example

Assuming #INFORMAT is set to TACL and your current IN file is your terminal, the following displays your terminal name:

```
67> #OUTPUT [#IN]
$DECAY
```

#INFORMAT Built-In Variable

Use #INFORMAT to set or obtain the current input format for the TACL IN file.

```
#INFORMAT
```

Result

#INFORMAT returns the current #INPUT formatting mode-PLAIN, QUOTED, or TACL.

Considerations

- #INFORMAT applies only to input read from the IN file, not to any other input data. The default format for macro or routine files and function libraries is TACL format; to change this setting, use a ?FORMAT directive.
- PLAIN format is always in effect for #REQUESTERs, #SERVERs, and #DELTA file operations, and for the IN option on #SET and SET.
- When you first log on, #INFORMAT is initialized to PLAIN.
- To change the default for your TACL session, use a #SET #IN statement. You can also specify the IN file when you start a TACL process.
- To read information from the current IN file, use the #INPUT built-in function.
- To set or obtain the current formatting mode for the TACL OUT file, use the #OUTFORMAT built-in variable.
- If #INFORMAT is TACL, any characters that are converted to internal format on input are stored with a tilde as a prefix. These characters include {, }, ==, [,], and |.
- #INFORMAT can affect how #ARGUMENT interprets arguments. For more information, refer to the description of #ARGUMENT Built-In Function on page 9-21.
- Use #PUSH #INFORMAT (or PUSH #INFORMAT) to save a copy of the current #INPUT formatting mode.
- Use #POP #INFORMAT (or POP #INFORMAT) to restore the last #INPUT formatting mode pushed.
- Use #SET #INFORMAT (or SET VARIABLE #INFORMAT) to establish the current formatting mode.

The syntax of #SET #INFORMAT is:

```
#SET #INFORMAT { PLAIN | QUOTED | TACL }
```

PLAIN

specifies that TACL should not translate characters read from IN, but should consider everything to be ordinary text; for example, braces and double equal signs are read as such and are not interpreted as comments in PLAIN mode.

QUOTED

causes TACL to translate metacharacters as if the formatting mode were TACL, but if text containing such characters is enclosed in quotation marks, TACL treats embedded metacharacters as if they were ordinary text (no tildes are needed). The #SET and #SETV built-in functions operate in different ways: #SET treats quotes as plain text; #SETV does not include the quotes (see Examples).

TACL

causes TACL to translate the metacharacters [and] (square brackets), { and } (braces), | (vertical line), == (double equals), and ~ (tilde) into internal notation. TACL reads square brackets as the beginning and ending of an invocation. A vertical line indicates a label in an enclosure. TACL reads braces and double equals as comments.

Using a tilde causes TACL to interpret the next character as plain text, rather than a delimiter; for example, TACL reads ~[as an ordinary open square bracket, rather than the beginning of an invocation. To use a tilde character as text, double it (~~). The tilde has no effect on its own, but only in conjunction with other characters.

When using the TACL format, you can put several commands on a single line by separating the commands with a tilde and a semicolon (~;). TACL translates these into internal end-of-line characters.

You can also use a tilde and an underscore (~_) when #INFORMAT is set to TACL. TACL translates this notation into an internal space. These metaspaces are printed as spaces if you use the PRETTY option with #OUTFORMAT; otherwise, they are treated as ordinary characters.

Examples

If you specify special characters in an IN file, #INFORMAT affects how TACL interprets the characters. This sends square brackets to TEDIT:

```
56> TEDIT critique; SEARCH [sic]
```

If the command is part of an IN file and #INFORMAT is set to TACL (or QUOTED), TACL tries to invoke SIC, considering it to be a variable, and displays an error message. If you change the command to

```
57> TEDIT critique; SEARCH ~[sic~]
```

TACL passes the brackets as text to TEDIT. Preceding any TACL metacharacter (including the tilde character itself) with a tilde causes TACL to consider it as simple text.

Table 9-11 lists the difference in text storage between TACL and QUOTED settings after the variables are initialized as follows (#OUTFORMAT is set to PLAIN):

```
#PUSH var b
#SET b $
```

Table 9-11. #INFORMAT Results

TACL Operation	#INFORMAT Set to TACL	#INFORMAT Set to QUOTED
#SET var a[b]c	a$c	a$c
#SET var “a[b]c”	“a$c”	“a[b]c”
#SET var “a~[b~]c”	“a[b]c”	“a~[b~]c”

These examples show differences in text storage between #SETV and #SET with the QUOTED #INFORMAT setting (#OUTFORMAT is set to PLAIN):

```
15> #PUSH var
16> #SET var “abc[def]ghi“
17> #OUTPUTV var
“abc[def]ghi“

18> #SETV var “abc[def]ghi“
19> #OUTPUTV var
abc[def]ghi

20>
```

#INITTERM Built-In Function

Use #INITTERM to reset your home terminal to its configured settings.

```
#INITTERM
```

Result

#INITTERM returns nothing.

Considerations

- You typically use this command when an application program leaves your terminal in an abnormal state.
- The #INITTERM built-in function calls the SETMODE operating system procedure, function 28. For information about setting device attributes, see the description of SETMODE setting for terminals in the *Guardian User's Guide* and the description of the SETMODE procedure in the *Guardian Procedure Calls Reference Manual*.

#INLINEECHO Built-In Variable

Use #INLINEECHO to determine whether TACL copies to its own OUT file the lines it passes to the IN file of an inline process.

```
#INLINEECHO
```

Result

#INLINEECHO returns 0 if echoing is disabled, -1 if it is enabled.

Considerations

- When you first log on, #INLINEECHO is initialized to zero.
- Use #PUSH #INLINEECHO (or PUSH #INLINEECHO) to save a copy of your current setting of the echo flag.
- Use #POP #INLINEECHO (or POP #INLINEECHO) to restore the echo flag from the last copy pushed.
- Use #SET #INLINEECHO (or SET VARIABLE #INLINEECHO) to enable or disable echoing of inline process input.

The syntax of #SET #INLINEECHO is:

```
#SET #INLINEECHO num
```

num

is zero to set the echo flag off, a nonzero value to set it on.

#INLINEEOF Built-In Function

Use #INLINEEOF to send an end-of-file character to a process under control of the INLINE facility.

```
#INLINEEOF
```

Result

#INLINEEOF returns nothing.

Considerations

- TACL waits until the inline process prompts for input then sends an EOF indication to that process.
- Use of #INLINEEOF when no inline process exists causes an error.

#INLINEOUT Built-In Variable

Use #INLINEOUT to control whether TACL copies to its own OUT file lines that inline processes write to their OUT files (when neither the OUT nor the OUTV option has been specified).

```
#INLINEOUT
```

Result

#INLINEOUT returns 0 if inline process output copying is disabled, -1 if it is enabled.

Considerations

- When you first log on, #INLINEOUT is initialized to -1.
- To permit OUT file copying, the inline process must not be started with the OUT or OUTV options.
- Use #PUSH #INLINEOUT (or PUSH #INLINEOUT) to save a copy of your current setting of the output copy flag.
- Use #POP #INLINEOUT (or POP #INLINEOUT) to restore the copy flag from the last version pushed.
- Use #SET #INLINEOUT (or SET VARIABLE #INLINEOUT) to enable or disable copying of inline process output.

The syntax of #SET #INLINEOUT is:

```
#SET #INLINEOUT num
```

num

is zero to set the copy flag off, a nonzero value to set it on.

#INLINEPREFIX Built-In Variable

Use #INLINEPREFIX to establish the prefix that identifies lines to be passed to inline processes. Lines that start with the defined prefix are not processed by TACL.

```
#INLINEPREFIX
```

Result

#INLINEPREFIX returns the character or characters that make up the current inline prefix, if any.

Considerations

- When using prefixed lines:
  - The prefix must be followed by a space unless it appears alone on a line. In that case, TACL passes a blank line to the inline process.
  - If a prefixed line appears when no inline process exists, an error occurs.
- Use #PUSH #INLINEPREFIX (or PUSH #INLINEPREFIX) to save a copy of your current inline prefix.
- Use #POP #INLINEPREFIX (or POP #INLINEPREFIX) to restore the inline prefix from the last copy pushed.
- Use #SET #INLINEPREFIX (or SET VARIABLE #INLINEPREFIX) to define an inline prefix.

  The syntax of #SET #INLINEPREFIX is:

  ```
  #SET #INLINEPREFIX [ prefix ]
  ```

  prefix

  is an optional one-character to four-character prefix. If you omit *prefix*, it is set to null.

  - The default prefix is a null value (no prefix). For efficiency, you should leave the prefix set to nothing when you are not using the INLINE facility.
  - You must not use “#SET” as the prefix; doing so prevents you from changing it to anything else.
  - The prefix cannot include space or end-of-line characters.
  - The prefix is not case-sensitive.
- For additional information and examples showing the use of #INLINEPREFIX, refer to the *TACL Programming Guide*.

#INLINEPROCESS Built-In Variable

Use #INLINEPROCESS to obtain the process ID of the current inline process.

```
#INLINEPROCESS
```

Result

#INLINEPROCESS returns the process ID of the current inline process, if one exists; if not, #INLINEPROCESS returns nothing.

Considerations

- Use #PUSH #INLINEPROCESS (or PUSH #INLINEPROCESS) to save the process ID of the current inline process. Pushing #INLINEPROCESS allows you to create a new inline process while another is still in existence. You cannot communicate with an inline process while it is pushed.
- Pushing and popping #INLINEPROCESS can help you avoid errors or timing problems (race conditions) if you stop an inline process and start another in quick succession. The first process may not stop as soon as you issue an EXIT command. If you push #INLINEPROCESS before starting the first process, then pop #INLINEPROCESS after issuing the EXIT command, TACL disassociates itself from that process without waiting for it to complete. You can then push #INLINEPROCESS again and start another process immediately.
- You cannot set #INLINEPROCESS. It is set implicitly by the creation and termination of inline processes.
- Use #POP #INLINEPROCESS (or POP #INLINEPROCESS) to break communication with the current inline process (if any) and reestablish communication with the inline process (if any) that was in effect when #INLINEPROCESS was last pushed.
- When terminating communication with an inline process, TACL waits until the process prompts for input, then sends an EOF indication to that process.
- Use of #INLINEEOF when no inline process exists results in an error. You can determine if there is a current INLINE process by testing as shown:

```
[#IF [#EMPTY [#INLINEPROCESS]] |THEN|
   == does not exist
|ELSE|
   == does exist
]
```

When you push #INLINEPROCESS, you can create another inline process only if your total number of #REQUESTERs, implicit and explicit #SERVERs, and inline processes is less than 100.

When you pop #INLINEPROCESS (or unframe it) and its corresponding inline process is still in existence, you can no longer communicate with that process;

- TACL responds to all its I/O requests with error 66. The process is still counted against the 100-process limit mentioned above, however, until it terminates.
- Pushing and popping #INLINEPROCESS has no effect on any of the other built-in variables related to inline processes (#INLINEECHO, #INLINEOUT, #INLINEPREFIX, or #INLINETO).
- When you first log on, #INLINEPROCESS is initialized to a null value.

#INLINETO Built-In Variable

Use #INLINETO to determine whether TACL copies to a variable level the lines that inline processes write to their OUT files (when neither the OUT nor the OUTV options have been specified).

```
#INLINETO
```

Result

#INLINETO returns the fully qualified name of the variable level to which inline process output is appended, if such a variable has been defined; otherwise, #INLINETO returns nothing.

Considerations

- When you first log on, #INLINETO is initialized to a null value.
- Use #PUSH #INLINETO (or PUSH #INLINETO) to save a copy of your current setting of the copy flag.
- Use #POP #INLINETO (or POP #INLINETO) to restore the identity of the output copy variable to that of the last one pushed.
- Use #SET #INLINETO (or SET VARIABLE #INLINETO) to establish a variable to which copies of inline process output are to be appended.

  The syntax of #SET #INLINETO is:

  ```
  #SET #INLINETO [ variable-level ]
  ```

  variable-level

  is the name of an existing variable level. It must not be a DIRECTORY, a STRUCT, or a STRUCT item. If you omit *variable-level*, you disable this copying feature.

#INPUT Built-In Function

Use #INPUT to read information from the TACL IN file.

```
#INPUT [ / option [ , option ] ... / ] [ prompt ]
```

option

is an option that qualifies the read operation. It can be any of these:

CURRENT

reads input from the current IN file (if #IN was pushed), rather than from the primary IN file.

FUNCTIONKEY *variable-level*

sets *variable-level* to the name of the function key used to terminate reading. If the RETURN key terminates reading, *variable-level* is cleared. Specify *variable-level* as the name of an existing variable level. When this option is in effect, text generated by a function key is removed from the input text.

HISTORY *history-prompt*

adds the line read to the history buffer. In addition, the history number and *history-prompt* are appended to prompt.

HISTORYV *variable-level*

adds the line read to the history buffer. In addition, the history number and the value of *variable-level* are appended to prompt.

NOECHO

suppresses echoing of the input line.

UNTIL { EOF | TACL }

specifies the conditions that terminate reading:

EOF

reads input until end-of-file is reached or, if you specify the FUNCTIONKEY option, until a function key is pressed.

TACL

reads input until all square brackets balance. While #INPUT is reading data, it processes function keys immediately and removes comments in the usual TACL manner.

If you omit the UNTIL option, pressing a function key or the RETURN key terminates reading. prompt is one or more characters to be written to the IN file as a prompt.

Consideration

TACL saves the primary IN file, set at startup time, as the default setting of the #IN built-in variable. If you set #IN to another file, TACL stores the second file as the current IN file. You can use the #INPUT built-in function to read from either file.

Result

#INPUT returns the line read from the input file.

Examples

This example illustrates the differences resulting from options in the reactions of #INPUT to function keys. Each of these read operations is terminated by function key 12:

1. With no options, #INPUT provides no carriage return, so the output in this example appears as a continuation of the input, though on the next line. Text supplied by the function key is included with the input text:

```
24> #OUTPUT [#INPUT ?]
?this
     thisK$%
```

2. Option processing provides a carriage return, so the output text appears at the beginning of the next line. With the FUNCTIONKEY option, the identity of the function key pressed is stored in the specified variable level:

```
25> #PUSH key
26> #OUTPUT [#INPUT /FUNCTIONKEY key/ ?] , [key]
?that
that , F12
```

3. With the UNTIL TACL option, the identity of the function key prefaces the input text:

```
27> #OUTPUT [#INPUT /UNTIL TACL/ ?]
?the_other
F12 the_other
```

4. These statements use #INPUT to prompt a user, save the name of the function key that terminated the input, and place the input into the history buffer:

```
29> #SET #OUTFORMAT PRETTY
30> #PUSH key text
31> #SET text [#INPUT / FUNCTIONKEY key, HISTORY ~_-->~_/]
32> --> this is a test == terminated with F12 key
33> OUTVAR key
F12
```

```
34> OUTVAR text
this is a test
35> HISTORY 4
```

```
32> this is a test
33> OUTVAR key
34> OUTVAR text
35> HISTORY 4
```

If you omit the HISTORY option in the #INPUT function call, "this is a test" does not appear in the history listing.

#INPUTEOF Built-In Variable

Use #INPUTEOF to obtain the current state of the end-of-file flag.

```
#INPUTEOF
```

Result

#INPUTEOF returns -1 if TACL reads an end-of-file; otherwise, it returns 0.

Considerations

- When you first log on, #INPUTEOF is initialized to zero.
- Every #INPUT or #INPUTV sets #INPUTEOF to 0. If TACL receives an end-of-file in response, it then sets #INPUTEOF to -1.
- Use #PUSH #INPUTEOF (or PUSH #INPUTEOF) to save a copy of the current setting of the INPUTEOF flag.
- Use #POP #INPUTEOF (or POP #INPUTEOF) to restore the INPUTEOF flag from the copy last pushed.
- Use #SET #INPUTEOF (or SET VARIABLE #INPUTEOF) to set the INPUTEOF flag to on or off.

  The syntax of #SET #INPUTEOF is:

  ```
  #SET #INPUTEOF num
  ```

  num

  is zero to set the flag off or a nonzero value to set the flag on.

#INPUTV Built-In Function

Use #INPUTV to read information from the TACL primary or current IN file (typically, the home terminal) into a variable level.

You can also use #INPUTV to prompt directly with the binary data of a STRUCT or to set the binary data of a STRUCT directly from the IN file. No type-checking is done on the input data.

```
#INPUTV [ / option [ , option ] ... / ] variable-level
  prompt-string
```

option

is an option that qualifies the read operation. The options available are the same as described for #INPUT with one addition: the operand of the HISTORYV option can be a string instead of merely a variable level.

variable-level

is the name of the variable level (which can be of type STRUCT) that is to hold the input data. If no data is read, *variable-level* is set to a blank line. The previous contents of the variable level are lost.

prompt-string

is the name of an existing variable level (not enclosed in square brackets), the first line of which is to be written to the IN file as a prompt, or text enclosed in quotation marks to be used for that purpose, or a concatenation of such entities.

The concatenation operator is '+' (the apostrophes must surround the plus character).

Result

#INPUTV returns nothing.

Considerations

- TACL sets #INPUTEOF to -1 if you type an end-of-file (CTRL-Y); otherwise, it is set to 0.
- If TACL receives an end-of-file, it clears *variable-level*.
- The form

  ```
  #INPUTV struct prompt
  ```

  is not equivalent to

  ```
  #SET struct [#INPUT [ prompt]]
  ```

The former construct is faster and is not subject to errors incurred in type-checking the input data.

- If the data from IN is longer than the data area of the STRUCT, the excess bytes are discarded without any notification.
- If the data from IN is shorter than the data area of the STRUCT, TACL sets the entire STRUCT to its default values and then moves the supplied data into the beginning of the data bytes of the STRUCT.

Example

This example shows the use of strings in both the HISTORYV operand and the prompt of an #INPUTV invocation.

```
#PUSH indata
#INPUTV /HISTORYV " : "/ indata "History line "
```

The resulting dialog at the IN file appears as:

```
History line 43 : user input
```

#INSPECT Built-In Variable

Use #INSPECT to set or obtain the current state of the inspect flag.

```
#INSPECT
```

Result

#INSPECT returns the current state of the inspect flag: OFF, ON, or SAVEABEND.

Considerations

- When you first log on, #INSPECT is initialized to OFF.
- Use #PUSH #INSPECT (or PUSH #INSPECT) to save a copy of the current setting of the inspect flag.
- Use #POP #INSPECT (or POP #INSPECT) to restore the inspect flag from the copy last pushed.
- Use #SET #INSPECT (or SET #INSPECT) to set the state of the inspect flag. The syntax of #SET #INSPECT is:

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

`OFF`

disables INSPECT and selects DEBUG as the default debugger. (DEBUG is the system default debugging utility.) DEBUG then prompts for input when any process created by the current TACL (or any of its descendants) enters the debug state.

`ON`

selects the INSPECT symbolic debugger as the default debugger for all programs started by the current TACL. INSPECT then prompts for input when any process created by the current TACL (or by any descendant of the current TACL) enters the DEBUG state.

`SAVEABEND`

establishes INSPECT as the default debugger and automatically creates a save file if the program ends abnormally.

H-Series Usage

The program DEBUG is not available for use on systems running H-series software.

The DEBUG command invokes a debugger, it can be Inspect, Native Inspect (eInspect, which is not in the family of Inspect debuggers), or Visual Inspect.

The rules about which debugger gets invoked are approximately the same as for the RUND command. That is, if the INSPECT attribute is set ON anywhere (in the object file during compilation, or on the TACL command line using the SET command), you will get a debugger in the Inspect family (either Inspect or VI), unless of course neither of these debuggers is available, and then you get the default debugger, eInspect. If the Inspect attribute is OFF, you get Native Inspect (eInspect).

Inspect is invoked only for TNS accelerated/interpreted programs (never for TNS/E native programs), while Visual Inspect can handle both of these. Native Inspect handles only TNS/E native programs and snapshots.

#INTERACTIVE Built-In Function

Use #INTERACTIVE to determine whether your TACL is interactive. TACL is considered to be interactive if its IN and OUT files are the same.

```
#INTERACTIVE [ / CURRENT / ]
```

Result

If you include the CURRENT option, #INTERACTIVE returns -1 if the current IN file is the same as the current OUT file (the contents of #IN are the same as the contents of #OUT); otherwise, it returns 0.

If you omit the CURRENT option, #INTERACTIVE returns -1 if the primary IN file is the same as the primary OUT file; otherwise, it returns 0.

The primary IN and OUT files are those in effect when TACL first starts (specified by the IN and OUT run options in the RUN command that initiates the TACL process). The current IN and OUT files may be different from those, if #SET #IN or #SET #OUT have been executed.

Consideration

TACL automatically determines whether it is interactive when it first starts to run.

#INTERPRETJULIANDAYNO Built-In Function

Use #INTERPRETJULIANDAYNO to convert a Julian day number to its corresponding calendar date. A Julian day number is an integer representing the number of days from the arbitrary date January 1, 4713 B.C. to the present day. Julian day numbers are included in four-word timestamps returned from the #JULIANTIMESTAMP function and from the CREATION_GMT and LASTOPEN_GMT options of the #FILEINFO function.

```
#INTERPRETJULIANDAYNO julian-day-num
```

julian-day-num

is the Julian day number to be interpreted.

Result

#INTERPRETJULIANDAYNO returns a space-separated list of the numeric year, month, and day.

Example

```
24> #OUTPUT [#INTERPRETJULIANDAYNO 2448047]
1990 6 4
```

#INTERPRETTIMESTAMP Built-In Function

Use #INTERPRETTIMESTAMP to convert a four-word timestamp to its date and time components.

```
#INTERPRETTIMESTAMP four-word-timestamp
```

four-word-timestamp

is the timestamp to be converted.

Result

#INTERPRETTIMESTAMP returns a space-separated list of nine numbers, consisting of the Julian day number, year, month, day, hour, minute, second, millisecond, and microsecond.

Example

This example illustrates #INTERPRETTIMESTAMP output:

```
21> #OUTPUT [#INTERPRETTIMESTAMP [#JULIANTIMESTAMP]]
2447701 1991 6 23 22 29 42 784 566
```

#INTERPRETTRANSID Built-In Function

Use #INTERPRETTRANSID to convert a numeric transaction ID into separate numeric values for the various components of a transaction ID.

```
#INTERPRETTRANSID transid
```

transid

is a numeric transaction ID.

Result

#INTERPRETTRANSID returns a numeric status code indicating the outcome of the conversion:

0 Successful conversion

-2 Invalid transaction ID

Any other value indicates a TACL problem and should be reported to your service provider.

If the numeric status is zero, it is followed by a space and a space-separated list of these transaction-ID components:

- System number
- CPU number
- Sequence number
- Crash count

Consideration

To convert the components of a transaction ID into a single numeric transaction ID, use the #COMPUTETRANSID built-in function.

#JULIANTIMESTAMP Built-In Function

Use #JULIANTIMESTAMP to obtain a selected four-word timestamp.

```
#JULIANTIMESTAMP [ type [ tuid-request ] ]
```

type

specifies the particular timestamp desired:

0 current GMT

1 GMT when the system was loaded

2 GMT when SYSGEN was run

If you omit type, 0 is assumed.

tuid-request

if not zero, specifies that the time update ID is to be included in the result. The default is zero.

Result

#JULIANTIMESTAMP returns the selected GMT timestamp; if *tuid-request* is not zero, the timestamp is followed by a space and the current time update ID.

Examples

This example shows #JULIANTIMESTAMP output:

```
22> #OUTPUT [JULIANTIMESTAMP]
2114814040705385 28
```

This example illustrates the use of four-word timestamps obtained from #JULIANTIMESTAMP and #FILEINFO to make a decision based on the age of a file (represented by the dummy argument %1%):

```
[#IF [#JULIANTIMESTAMP] - [#FILEINFO /CREATION_GMT/.%1%]
    > 315360000000000 == Julian number equivalent to 1 year
|THEN|
  #OUTPUT %1% is more than 1 year old. Purge it?
]
```

#KEEP Built-In Function

Use #KEEP to remove all but the top n definition levels of one or more variables.

```
#KEEP num variable
```

num

is the number of levels to keep.

variable

is the name of an existing variable.

Result

#KEEP returns as its result the number of levels deleted.

Consideration

#KEEP 0 removes the variable.

#KEYS Built-In Function

Use #KEYS to determine which function keys are currently defined.

```
#KEYS
```

Result

#KEYS returns a space-separated list containing the names of all defined function keys.

Example

Assuming #INFORMAT is set to TACL, this statement displays the currently defined keys:

```
14> #OUTPUT [#KEYS]
F1 F2 F3 F4 F5 F6 F7 F8 F9 F10 F11 F12 F13 F14 F15 SF1 SF16
```

#LINEADDR Built-In Function

Use #LINEADDR to convert a character address to a line address.

```
#LINEADDR variable-level char-addr
```

variable-level

is an existing variable level to be operated upon. It must not be a DIRECTORY, a STRUCT, or a STRUCT item.

char-addr

is an integer or a variable level that contains an integer. *char-addr* specifies the character address of the character to be converted. The character address must be in the range from 1 to *max-int*, inclusive.

Result

#LINEADDR returns the line number of the line that contains the specified character.

Considerations

- If *char-addr* is greater than the number of characters in the variable level, #LINEADDR returns a value one greater than the number of the last line.
- If *variable-level* is empty, #LINEADDR returns 0.
- Each logical line contains an end-of-line character that counts as one byte. For variables that contain TACL statements, each metacharacter uses the number of visible characters plus one, including unprintable characters that are subject to change from one TACL RVU to another. Nonmetacharacters use one byte.

Example

Assume that var is a variable level containing:

```
ABCDEFG
HIJKLMNOPQRST
UVWXYZ
```

The invocation:

```
#LINEADDR var 10
```

returns 2; the tenth character in var, counting the end-of-line between lines 1 and 2 as one character, is the “I” in line 2.

#LINEBREAK Built-In Function

Use #LINEBREAK to insert an end-of-line into a variable level at a specified point in a specified line.

```
#LINEBREAK variable-level line-addr char-offset
```

variable-level

is the name of an existing variable level into which the end-of-line is to be inserted. It must not be in a shared segment and must not be a DIRECTORY, a STRUCT, or a STRUCT item.

line-addr

is an integer or a variable level that contains an integer. *line-addr* specifies the line number into which the end-of-line is to be inserted. The line address must be in the range from 1 to *max-int*, inclusive.

char-offset

is an integer or a variable level that contains an integer. *char-offset* is the position of the character within the specified line at which the end-of-line is to be inserted.

Result

#LINEBREAK returns nothing.

Considerations

- Each logical line contains an internal end-of-line character that counts as one byte.When you use the #LINEBREAK built-in function, TACL inserts the end-of-line character immediately before the specified character.
- If *char-offset* is greater than the number of characters in the line, the new end-of- line is inserted immediately before the existing end-of-line.
- If *line-addr* and*char-offset* are both 1, an end-of-line is inserted at the beginning of the variable level. If *line-addr* is beyond the end of the variable level, no end-of-line is inserted.

Example

Assume that var is a variable level containing:

```
ABCDEFG
HIJKLMNOPQRST
UVWXYZ
```

The invocation:

```
#LINEBREAK var 2 5
```

causes var to contain:

```
ABCDEFG
HIJK
LMNOPQRST
UVWXYZ
```

#LINECOUNT Built-In unction

Use #LINECOUNT to obtain the number of lines in a variable level.

```
#LINECOUNT variable-level
```

variable-level

is the name of an existing variable level whose lines are to be counted. It must not be a DIRECTORY, a STRUCT, or a STRUCT item.

Result

#LINECOUNT returns the number of lines in the variable level.

Example

Assume that var is a variable level containing:

```
ABCDEFG
HIJKLMNOPQRST
UVWXYZ
```

The invocation:

```
#LINECOUNT var
```

returns 3.

#LINEDEL Built-In Function

Use #LINEDEL to delete lines from a variable level, starting at a specified line.

```
#LINEDEL variable-level line-addr-1
  [ [ FOR line-count ] | TO line-addr-2 ] ]
```

variable-level

is an existing variable level from which lines are to be deleted. It must not be in a shared segment, and must not be a DIRECTORY, a STRUCT, or a STRUCT item.

line-addr-1

is an integer or a variable level that contains an integer. *line-addr-1* specifies the line number at which line deletion is to begin. The line address must be in the range from 1 to *max-int*, inclusive.

line-count

is an integer or a variable level that contains an integer. *line-count* specifies the number of lines to be deleted. The line count must be in the range from 0 to *max-int*, inclusive.

line-addr-2

is an integer or a variable level that contains an integer. *line-addr-2* specifies the line number at which line deletion is to end. The line address must be in the range from 0 to *max-int*, inclusive.

Result

#LINEDEL returns nothing.

Considerations

- If you use TO, the line specified by *line-addr-2* is included in the deletion: That is, “x TO y” is equivalent to “x FOR (y-x)+1.”
- If you use TO and *line-addr-1* is greater than *line-addr-2*, no deletion occurs.
- If you omit both FOR and TO, TACL deletes the line specified by *line-addr-1*.

Examples

Any part of the specified deletion that lies beyond the end of the variable level is ignored.

Assume that var is a variable level containing:

```
THE QUICK BROWN
FOX JUMPED OVER
THE LAZY DOG
TWICE A DAY
EXCEPT TUESDAYS.
```

1. Either of the invocations:

   `#LINEDEL var 2 TO 4` or `#LINEDEL var 2 FOR 3`

   causes var to contain:

   ```
   THE QUICK BROWN
   EXCEPT TUESDAYS.
   ```

2. Either of the invocations:

   `#LINEDEL var 2 TO 50` or `#LINEDEL var 2 FOR 49`

   causes var to contain:

   ```
   THE QUICK BROWN
   ```

#LINEFIND Built-In Function

Use #LINEFIND to find text in a variable level, searching forward from a specified line number.

```
#LINEFIND [ / EXACT / ] variable-level line-addr text
```

EXACT

specifies that the search is to be case-sensitive; if you omit it, the search makes no distinction between uppercase and lowercase letters.

variable-level

is an existing variable level within which TACL will search for text. It must not be a DIRECTORY, a STRUCT, or a STRUCT item.

line-addr

is an integer or a variable level that contains an integer. *line-addr* specifies the line number at which the search is to begin. The line address must be in the range from 1 to *max-int*, inclusive.

text

is the text constant to be found. The largest valid text length is 32,000 words minus the current contents of the stack. The amount of remaining space is typically 25,000 words long.

Result

#LINEFIND returns the number of the line in which text begins. If text is not found, or if text is empty, #LINEFIND returns zero.

Considerations

- If *line-addr* is past the end of the variable level, #LINEFIND returns zero.
- A text specification can include internal end-of-line characters if the entire invocation is enclosed in square brackets, but leading and trailing spaces and end-of- lines are ignored.
- The search begins immediately at the line number specified. If you make repeated calls to this function, using the result of each as a starting point for the next, you must add 1 to that result before supplying it to a subsequent call.
- If *line-addr* is empty, #LINEFIND returns zero.

Examples

Assume that var is a variable level containing:

```
ABCDEFG
HIJKLMNOPQRST
UVWXYZ
```

1. The invocation:

   ```
   #LINEFIND var 1 IJK
   ```

   returns 2; the first occurrence of IJK starting in or after line 1 is in line number 2.

2. The invocation:

   ```
   #LINEFIND var 2 IJK
   ```

   returns 2; the first occurrence of IJK is in the specified starting line, line 2.

3. The invocation:

   ```
   #LINEFIND var 3 IJK
   ```

   returns 0; there are no occurrences of IJK starting in or after line 3.

4. The invocation:

   ```
   #LINEFIND var 1 FOO
   ```

   returns 0; there are no occurrences of FOO anywhere in var.

#LINEFINDR Built-In Function

Use #LINEFINDR to find text in a variable level, searching backward from a specified line number.

```
#LINEFINDR [ / EXACT / ] variable-level line-addr text
```

EXACT

specifies that the search is to be case-sensitive; if you omit it, the search makes no distinction between uppercase and lowercase letters.

variable-level

is an existing variable level within which TACL will search for text. It must not be a DIRECTORY, a STRUCT, or a STRUCT item.

line-addr

is an integer or a variable level that contains an integer. *line-addr* specifies the line number at which the search is to begin. The line address must be in the range from 1 to *max-int*, inclusive. The search moves backward from this point.

text

is the text constant to be found. The largest valid text length is 32,000 words minus the current contents of the stack. The amount of remaining space is typically 25,000 words long.

Result

#LINEFINDR returns the number of the line in which text begins. If text is not found or text is empty, #LINEFINDR returns zero.

Considerations

- If *line-addr* is past the end of the variable level, #LINEFINDR starts the search at the end of the contents of the variable.
- A text specification can include internal end-of-line characters if the entire invocation is enclosed in square brackets, but leading and trailing spaces and end-of- lines are ignored.
- The search includes all of the specified line. The search finds a match if the matching text begins anywhere within, or before, that line (even if the text extends beyond the line). If you make repeated calls to this function, using the result of one call as a starting point for the next, you must subtract one from that result before using it in a subsequent call. This procedure avoids finding the same text again.

Examples

Assume that var is a variable level containing:

```
ABCDEFG
HIJKLMNOPQRST
UVWXYZ
```

1. The invocation:

   ```
   #LINEFINDR var 3 IJK
   ```

   returns 2; the nearest occurrence of IJK beginning in or before line 3 is in line 2.

2. The invocation:

   ```
   [#LINEFINDR var 2 T
   UV]
   ```

   returns 2; though the text T(end-of-line)UV carries over to line 3, it begins in line 2.

3. The invocation:

   ```
   #LINEFINDR var 4 FOO
   ```

   returns 0; there are no occurrences of FOO anywhere in var.

#LINEFINDRV Built-In Function

Use #LINEFINDRV to find a string in a variable level, searching backward from a specified line.

```
#LINEFINDRV [ / EXACT / ] variable-level line-addr string
```

EXACT

specifies that the search is to be case-sensitive; if you omit it, the search makes no distinction between uppercase and lowercase letters.

variable-level

is an existing variable level within which TACL will search for string. It must not be a DIRECTORY, a STRUCT, or a STRUCT item.

line-addr

is an integer or a variable level that contains an integer. *line-addr* specifies the line number at which the search is to begin. The line address must be in the range from 1 to *max-int*, inclusive. The search moves backward from this point.

string

is the string constant or the name of a variable level that contains text. string specifies the characters to be found. It must not be in a shared segment or be a DIRECTORY, a STRUCT, or a STRUCT item. The largest valid string length is 32,000 words minus the current contents of the stack. The amount of remaining space is typically 25,000 words long.

Result

#LINEFINDRV returns the number of the line in which string begins. If string is not found, or if string is empty, #LINEFINDRV returns zero. If *variable-level* is empty, then #LINEFINDRV returns zero.

Considerations

- If *line-addr* is past the end of the variable level, #LINEFINDRV starts the search at the end of the contents of the variable.
- One trailing end-of-line in string is ignored. Leading and trailing spaces are preserved, as are all other end-of-lines.
- The search includes all of the specified line. The search finds a match if the matching string begins anywhere within, or before, that line (even if the string extends beyond the line). If you make repeated calls to this function, using the result of one call as a starting point for the next, you must subtract one from that

result before using it in a subsequent call. This procedure avoids finding the same string again.

- Each line break contains an internal end-of-line character that counts as one byte.
- For variables that contain TACL statements, each square bracket ([,]), vertical bar (|), or tilde-space combination (~_) uses two bytes, including unprintable characters that are subject to change from one TACL RVU to another. Other characters use one byte.

Examples

Assume that var is a variable level containing:

```
ABCDEFG
HIJKLMNOPQRST
UVWXYZ
```

and that var2 is a variable level containing:

```
IJK
```

1. Either of the invocations:

   ```
   #LINEFINDRV var 3 "IJK"
   ```
   or
   ```
   #LINEFINDRV var 3 var2
   ```

   returns 2; the nearest occurrence of IJK beginning in or before line 3 starts in line 2.

2. The invocation:

   ```
   #LINEFINDRV var 2 "UVW"
   ```

   returns 0; there are no occurrences of UVW beginning in or before line 2.

#LINEFINDV Built-In Function

Use #LINEFINDV to find a string in a variable level, searching forward from a specified line.

```
#LINEFINDV [ / EXACT / ] variable-level line-addr string
```

EXACT

specifies that the search is to be case-sensitive; if you omit it, the search makes no distinction between uppercase and lowercase letters.

variable-level

is an existing variable level in which TACL will search for string. It must not be a DIRECTORY, a STRUCT, or a STRUCT item.

line-addr

is an integer or a variable level that contains an integer. *line-addr* specifies the line number at which the search is to begin. The line address must be in the range from 1 to *max-int*, inclusive.

string

is the string constant or the name of a variable level that contains text. string specifies the characters to be found. It must not be in a shared segment or be a DIRECTORY, a STRUCT, or a STRUCT item. The largest valid string length is 32,000 words minus the current contents of the stack. The amount of remaining space is typically 25,000 words long.

Result

#LINEFINDV returns the number of the line in which string begins. If string is not found or the string is empty, #LINEFINDV returns zero. If *variable-level* is empty, then #LINEFINDV returns zero.

Considerations

If *line-addr* is past the end of the variable level, #LINEFINDV returns zero. The search starts immediately at the beginning of the line specified. If you make repeated calls to this function, using the result of each as a starting point for the next, you must add 1 to that result before supplying it to a subsequent call.

Examples

Assume that var is a variable level containing:

```
ABCDEFG
HIJKLMNOPQRST
UVWXYZ
```

and that var2 is a variable level containing:

```
IJK
```

1. Either of the invocations:

   `#LINEFINDV var 1 "IJK"` or `#LINEFINDV var 1 var2`

   returns 2; the first occurrence of IJK starting in or after line 1 is in line 2.

2. Either of the invocations:

   `#LINEFINDV var 2 "IJK"` or `#LINEFINDV var 2 var2`

   returns 2; the first occurrence of IJK is exactly at the starting point, the beginning of line 2.

3. Either of the invocations:

   `#LINEFINDV var 3 "IJK"` or `#LINEFINDV var 3 var2`

   returns 0; there are no occurrences of IJK starting in or after line 3.

4. The invocation:

   `#LINEFINDV var 1 "FOO"`

   returns 0; there are no occurrences of FOO anywhere in var.

#LINEGET Built-In Function

Use #LINEGET to obtain a copy of a set of consecutive lines in a variable level.

```
#LINEGET string line-addr-1
   { { FOR line-count } | { TO line-addr-2 } }
```

string

is an existing variable level or quoted text from which lines are to be copied. It must not be a DIRECTORY, a STRUCT, or a STRUCT item.

line-addr-1

is an integer or a variable level that contains an integer. *line-addr-1* specifies the line number at which copying is to begin. The line address must be in the range from 1 to *max-int*, inclusive.

line-count

is an integer or a variable level that contains an integer. *line-count* specifies the number of lines to copy. The line count must be in the range from 0 to *max-int*, inclusive.

line-addr-2

is an integer or a variable level that contains an integer. *line-addr-2* specifies the line number at which copying is to end. The line address must be in the range from 1 to *max-int*, inclusive.

Result

#LINEGET returns as its result the copied lines.

Considerations

- If you use TO, the line specified by *line-addr-2* is included in the copy; that is, “x TO y” is equivalent to “x FOR (y-x)+1.”
- If you use TO and *line-addr-1* is greater than or equal to *line-addr-2*, or if you use FOR and *line-count* is zero, no copying occurs.
- If you omit both FOR and TO, one line is copied.
- Any part of the specified copy that lies beyond the end of the variable level is ignored.
- If #LINEGET is to return more than one line, you must enclose in square brackets the invocation of the function that obtains that result.

Examples

Assume that var is a variable level containing:

```
THE QUICK BROWN
FOX JUMPED OVER
THE LAZY DOG
TWICE A DAY
EXCEPT TUESDAYS.
```

1. The invocation:

   ```
   #LINEGET var 2
   ```

   returns:

   ```
   FOX JUMPED OVER
   ```

2. Either of the invocations:

   ```
   #LINEGET var 2 TO 4 or #LINEGET var 2 FOR 3
   ```

   returns:

   ```
   FOX JUMPED OVER
   THE LAZY DOG
   TWICE A DAY
   ```

   Therefore, a function invocation producing that result must be enclosed in square brackets:

   ```
   [#OUTPUT [#LINEGET var 2 FOR 3]]
   ```

#LINEGETV Built-In Function

Use #LINEGETV to copy a set of consecutive lines from one variable level to another.

```
#LINEGETV string variable-level line-addr-1
  [ [ FOR line-count ] | [ TO line-addr-2 ] ]
```

string

is an existing variable level or quoted text from which characters are to be copied. It must not be a DIRECTORY, a STRUCT, or a STRUCT item.

variable-level

is an existing variable level that is to receive the copy. The variable's original contents are lost, and its type is set to TEXT.

line-addr-1

is an integer or a variable level that contains an integer. *line-addr-1* specifies the line number at which copying is to begin. The line address must be in the range from 1 to *max-int*, inclusive.

line-count

is an integer or a variable level that contains an integer. *line-count* specifies the the number of lines to copy. The line count must be in the range from 0 to *max-int*, inclusive.

line-addr-2

is an integer or a variable level that contains an integer. *line-addr-1* specifies the line number at which copying is to end. The line address must be in the range from 0 to *max-int*, inclusive.

Result

#LINEGETV returns nothing.

Considerations

- If you use TO, the line specified by *line-addr-2* is included in the copy: That is, "x TO y" is equivalent to "x FOR (y-x)+1."
- If you use TO and *line-addr-1* is greater than or equal to *line-addr-2*, or if you use FOR and *line-count*is less than one, no copying occurs.
- If you omit both FOR and TO, one line is copied.
- Any part of the specified copy that lies beyond the end of string is ignored.

Examples

Assume that var is a variable level containing:

```
THE QUICK BROWN
FOX JUMPED OVER
THE LAZY DOG
TWICE A DAY
EXCEPT TUESDAYS.
```

Either of the invocations:

```
#LINEGETV var var2 2 TO 3
```
or
```
#LINEGETV var var2 2 FOR 4
```

set var2 to:

```
FOX JUMPED OVER
THE LAZY DOG
```

The invocation:

```
#LINEGETV var var2 3
```

sets var2 to:

```
THE LAZY DOG
```

#LINEINS Built-In Function

Use #LINEINS to insert text into a variable level at a specified line number.

```
#LINEINS variable-level line-addr text
```

variable-level

is an existing variable level into which text will be inserted. It must not be in a shared segment, and must not be a DIRECTORY, a STRUCT, or a STRUCT item.

line-addr

is an integer or a variable level that contains an integer. *line-addr* specifies the line number at which text is to be inserted. The line address must be in the range from 1 to *max-int*, inclusive.

text

is the text constant to be inserted. The largest valid text length is 32,000 words minus the current contents of the stack. The amount of remaining space is typically 25,000 words long.

Result

#LINEINS returns nothing.

Considerations

- A text specification can include internal end-of-lines if you enclose the entire invocation in square brackets, but leading and trailing spaces and end-of-lines are ignored.
- If *line-addr*r is beyond the end of the variable level, the text is appended to the end of the variable level, starting on a new line.

Examples

Assume that var is a variable level containing:

```
ABCDEFG
HIJKLMNOPQRST
UVWXYZ
```

1. The invocation:

   ```
   #LINEINS var 3 NEW TEXT
   ```

   causes var to contain:

   ```
   ABCDEFG
   HIJKLMNOPQRST
   ```

```
NEW TEXT
UVWXYZ
```

2. The invocation:

```
[#LINEINS var 3 NEW
TEXT]
```

causes var to contain:

```
ABCDEFG
HIJKLMNOPQRST
NEW
TEXT
UVWXYZ
```

3. The invocation:

```
#LINEINS var 100 NEW TEXT
```

causes var to contain:

```
ABCDEFG
HIJKLMNOPQRST
UVWXYZ
NEW TEXT
```

#LINEINSV Built-In Function

Use #LINEINSV to insert a string into a variable level at a specified line number.

```
#LINEINSV variable-level line-addr string
```

variable-level

is an existing variable level into which string will be inserted. It must not be in a shared segment and must not be a DIRECTORY, a STRUCT, or a STRUCT item.

line-addr

is an integer or a variable level that contains an integer. *line-addr* specifies the line number at which text is to be inserted. The line address must be in the range from 1 to *max-int*, inclusive.

string

is the string constant or the name of a variable level that contains text to be inserted. string specifies the characters to be found. It must not be in a shared segment or be a DIRECTORY, a STRUCT, or a STRUCT item. The largest valid string length is 32,000 words minus the current contents of the stack. The amount of remaining space is typically 25,000 words long.

Result

#LINEINSV returns nothing.

Considerations

- Leading and trailing spaces in string are preserved, as are all end-of-lines.
- If *line-addr* is beyond the end of the variable level, string is appended to the end of the variable level, starting on a new line.

Examples

Assume that var is a variable level containing:

```
ABCDEFG
HIJKLMNOPQRST
UVWXYZ
```

and that var2 is a variable level containing:

```
NEW
TEXT
```

1. The invocation:

```
#LINEINSV var 3 var2
```

causes var to contain:

```
ABCDEFG
HIJKLMNOPQRST
NEW
TEXT
UVWXYZ
```

2. The invocation:

```
#LINEINSV var 100 "NEW TEXT"
```

causes var to contain:

```
ABCDEFG
HIJKLMNOPQRST
UVWXYZ
NEW TEXT
```

#LINEJOIN Built-In Function

Use #LINEJOIN to delete the end-of-line at the end of a specified line, thereby joining the following line to it.

```
#LINEJOIN variable-level line-addr
```

variable-level

is an existing variable level in which joining is to take place.

line-addr

is an integer or a variable level that contains an integer. *line-add* specifies the line number from which the end-of-line is to be deleted. The line address must be in the range from 1 to *max-int*, inclusive.

Result

#LINEJOIN returns nothing.

Considerations

- If *line-addr* is equal to or greater than the number of the last line, no joining occurs.
- Lines are joined with no intervening new character.
- You can use #CHARDEL to delete an end-of-line at a specific character position.

Examples

Assume that var is a variable level containing this:

```
ABCDEFG
HIJKLMNOPQRST
UVWXYZ
```

The invocation:

```
#LINEJOIN var 2
```

causes var to contain:

```
ABCDEFG
HIJKLMNOPQRSTUVWXYZ
```

#LOAD Built-In Function

Use #LOAD to load variables from a TACL library file into memory. A library file consists of one or more ?SECTION directives of the form:

```
?SECTION name type
```

Each directive is followed by one or more lines of text that are to be associated with the variable level specified by name. The nature of the text is largely dependent on the type of the section, which can be ALIAS, DELTA, MACRO, ROUTINE, STRUCT, or TEXT. The body of text ends with the next ?SECTION directive, or the end of the file, whichever comes first.

```
#LOAD [ / option [ , option ] / ] file-name
```

option

can be either of these:

KEEP *num*

causes TACL to perform an implicit KEEP num command on each variable it loads.

LOADED *variable-level*

causes the list of variables loaded to be placed in *variable-level* rather than in the result of #LOAD; *variable-level* must already exist. Its original contents are lost and its type is set to TEXT. *variable-level* contains one variable name per line, so it is suitable for processing by #EXTRACT(V).

file-name

is the name of a TACL library file.

Result

#LOAD returns a space-separated list containing the names of the variables loaded. If you specify the LOADED option, however, #LOAD returns nothing.

Considerations

- The LOADED option is especially useful for loading libraries containing large numbers of variables (so many that a “text buffer overflow” error results when they are placed in the result of #LOAD).
- If you need to include a blank line (often useful in DELTA type variables), use the ?BLANK directive. ?BLANK causes the loader to insert a blank line in the variable level.

- To include lines beginning with question marks, (for example, you might be loading DDL commands into a variable level for later use as the IN variable for a DDL run), double the question mark (??). The first question mark and any spaces adjacent to it are discarded and the remainder of the line is treated as text.
- For each ?SECTION name type directive in a library file, TACL pushes a variable named name, sets its type to type, and sets its contents to text (all the text that follows the ?SECTION directive, until the next directive or the end of the library file). If the variable name already exists, TACL pushes the variable and puts text in the new top level.
- #LOAD reads data from a library file in TACL format unless the file contains a ?FORMAT PLAIN or ?FORMAT QUOTED directive to specify otherwise. If changed, the format reverts to TACL at the next ?SECTION directive.
- The definition of a STRUCT in a library file cannot contain square brackets; the body of the structure must not be on the same line as the ?SECTION directive.
- If you load a variable that is already loaded, TACL pushes the variable and creates a new variable level.
- To obtain a list of variables loaded into your home directory, use the VARIABLES command or the #VARIABLES built-in function.
- To obtain detailed information about a variable in your home directory, use the VARINFO command or the VARIABLEINFO built-in function.
- To delete a variable, use the KEEP command, the #KEEP built-in function, or the KEEP option of the #LOAD built-in function (to delete and load a variable simultaneously):

  ```
  #SET rslt [#LOAD /KEEP 1/ TESTS]
  ```

Examples

Following are some examples of ?SECTION directives in library files that can be processed by #LOAD:

```
?SECTION versnum TEXT
== Version of this library
14OCT91

?SECTION setmacro MACRO
== Creates a macro (%1%) defined by text (%2 to *%)
[#DEF %1% MACRO |BODY|%2 to *%]

?SECTION setvar ALIAS
== Defines a variable
SETMACRO

?SECTION nocommas MACRO
== Turns comma-separated list into space-separated list
[#DELTA /COMMANDS nocommas_d/ %*%]

?SECTION nocommas_d DELTA
== Engine for NOCOMMAS
J<:S,$;-DI $>

?SECTION inventory STRUCT
  BEGIN
    INT item;
    INT price;
    INT quantity;
  END;

?SECTION obsolete^items STRUCT
  LIKE inventory;
```

#LOCKINFO Built-In Function

Use #LOCKINFO to obtain information about record locks.

```
#LOCKINFO lock-spec tag buffer
```

lock-spec

consists of a numeral followed by one or more descriptive fields, and specifies the type of lock information wanted; *lock-spec* can be any of these:

`0` *device-name*

searches the specified device for all locks.

`1` *file-name*

searches the specified file for all locks.

`2` *device-name* [*node-name*.] {$*process-name* | *cpu,pin* }

searches the specified device for all locks set by the specified process.

`3` *device-name transid*

searches the specified device for all locks under the specified transaction ID. *transid* is a four-word transaction identifier that uniquely identifies a specific transaction protected by the TMF subsystem. *transid* must have this format:

node (*crash-count*). *cpu*. *sequence-number*

(*crash-count*) can be omitted if it equals 0. \ *node* can specify a node name or node number and can be omitted for a local node if *crash-count* is also omitted.

tag

is a number identifying a specific set of lock information. If tag is zero, #LOCKINFO obtains the first lock information. #LOCKINFO includes in its result the tag value you need to get the next lock information in sequence.

buffer

is a variable level that is to receive the lock information; it must be a writable STRUCT (see “Example” for a suggested structure definition). The minimum length for buffer is 294 bytes.

Result

#LOCKINFO returns a file-system error code indicating the outcome of the operation. If the code is 0 (no error) or 45 (file is full-no error otherwise), it is followed by a space and the tag to be passed in the next call to #LOCKINFO.

Consideration

For D-series programs, use the #FILEGETLOCKINFO built-in function.

Example

This routine accepts a volume name and reports all locks for that volume.

```
?SECTION looklock ROUTINE
#FRAME
== Define buffer into which #LOCKINFO puts lock information.
[#DEF buffer STRUCT
  BEGIN
  STRUCT lib;
    BEGIN
    BYTE type;
    BYTE keylen;
    INT misc;
    CHAR svol^file(0:15);
    INT numlab;
    INT2 laboff; == byte offset of 1st labinfo entry
    BYTE keyval(0:255);
    INT2 recaddr REDEFINES keyval;
    == Be sure there is space for at least 1 labinfo entry.
    == More might fit if keylen < 256.
    FILLER 12;
    END; == lib
  STRUCT generic REDEFINES lib;
    BEGIN
    BYTE onechar (0:293);
    END;
  END; == buffer
] == end #DEF

== Define structure for one labinfo entry.
== One entry at a time is copied here from buffer.
[#DEF labinfo STRUCT
  BEGIN
  INT misc;
  CRTPID locker;
  TRANSID translocker REDEFINES locker;
  INT reserved;
  END; == labinfo
] == end #DEF

== Get byte length of labinfo STRUCT
#PUSH labinfolen
#SET labinfolen [#VARIABLEINFO /LEN/ labinfo]

== Define macro to show information about one locked &
   resource (a file or a record).
[#DEF display_lib MACRO |BODY|
  #OUTPUT
  == Get interesting values from buffer.
  #SET type [buffer:lib:type]
```

```
  #SET keylen [buffer:lib:keylen]
  #SET fname [vol].[buffer:lib:svol^file(0:7)]
  #SET fname [fname].[buffer:lib:svol^file(8:15)]
  #SET numlab [buffer:lib:numlab]
  #SET laboff [buffer:lib:laboff]

== Display them.
#OUTPUT type: [buffer:lib:type]
#OUTPUT keylen: [keylen]
#OUTPUT laboff: [laboff]
== Is it a record lock or a file lock?
[#IF ([buffer:lib:type]) |THEN|
  #OUTPUT RECORD LOCK on [fname]
  [#IF (keylen = 0) |THEN|
    #OUTPUT ID: [buffer:lib:recaddr]
  |ELSE|
    #OUTPUT ID: [buffer:lib:keyval(0:[#compute (keylen-1)])]
  ]
|ELSE|
  #OUTPUT FILE LOCK on [fname]
]

== Show number of lockers/waiters.
#OUTPUT [numlab] locks and waits for [fname]
] == end DISPLAY_LIB macro

== Define macro to display all labinfo entries returned for &
   one locked resource (a file or a record).
[#DEF display_lab MACRO |BODY|
  == Loop for each labinfo entry. There are [numlab] entries.
  #SET count 0
 [#LOOP |WHILE| (count < numlab) |DO|
  == Compute location of current labinfo entry.
  #SET start [#COMPUTE laboff + (labinfolen * count)]
  #SET stop [#COMPUTE start + labinfolen-1]

  == Copy entry from buffer to labinfo structure.
  #SETBYTES labinfo buffer:generic:onechar([start]:[stop])

  == Display entry.
  #OUTPUTV labinfo

  == Increment entry number.
  #SET count [#COMPUTE count + 1]

  ] == end #LOOP
] == end DISPLAY_LAB macro

== Initialize variables
#PUSH vol err tag
#PUSH type keylen fname numlab laboff
```

```
#PUSH count start stop
#SET tag 0

== Get name of volume to be searched for locks.
SINK [#ARGUMENT /VALUE vol/ DEVICE]

== Loop through all locks on volume.
[#LOOP |DO|

  == Get lock info.
  #SETMANY err tag, [#LOCKINFO 0 [vol] [tag] buffer]

  == Check if locks found.
  [#IF (( err = 0) OR (err = 45 )) |THEN|
    == Display lib info for this resource.
    DISPLAY_LIB
    == Display all labinfo entries obtained on this call &
      to #LOCKINFO.
    DISPLAY_LAB

    == More lockers/waiters for this same resource?
    [#IF (err=45) |THEN|
      #OUTPUT There were more lockers/waiters for this
      #OUTPUT resource than would fit in the buffer.
    ]
  ] == Locks found.
  |UNTIL| (( err <> 0) AND (err <> 45))
] == end #LOOP

== Error 1 is normal termination. Otherwise:
[#IF (err <> 1) |THEN|
   #OUTPUT Error [err] on [vol]
]
#UNFRAME
```

Additional information is available in the *Guardian Procedure Calls Reference Manual*.

#LOGOFF Built-In Function

Use #LOGOFF to log off the current TACL.

```
#LOGOFF [ / option [ , option ] ... / ]
```

option

is any of these:

CLEAR

clears the terminal screen after you log off (for use when TACL is not configured to clear the screen automatically).

NOCLEAR

prevents TACL from clearing the terminal screen after you log off (for use when TACL is configured to clear the screen automatically-the usual case).

PAUSE

causes TACL to execute a PAUSE command immediately following the LOGOFF command.

SEGRELEASE

immediately releases the extended segment used to hold your variables. Typically, TACL saves this segment to use in case you are the next user of this TACL, but if you fill up your variable space and processing cannot proceed, you can log off using the SEGRELEASE option to discard the variable space.

Result

#LOGOFF returns nothing.

Considerations

- When you log off, any processes that you started continue to run.
- Any macro or routine running at the time #LOGOFF occurs terminates immediately. A subsequent LOGON does not restart it. For example:

  ```
  #FRAME
  ...
  #LOGOFF
  #UNFRAME
  ```

  causes the frame counter to remain in its incremented condition because #UNFRAME is not executed.

- If you use the #LOGOFF function while working through a modem, the modem disconnects (unless the ancestor of the TACL process is running in another system).
- If the ancestor of your current TACL process is a process running in another system and you enter the LOGOFF command, the current TACL process is deleted and control returns to the ancestor process. This message is displayed:

  `Exiting from TACL on \`*`node-name`*
- If you are accessing a remote node through a modem on your local node, TACL does not issue a modem disconnect.
- Any process that tries to use your variables after you log off receives error 66 on its I/O requests.
- If your TACL is interactive and you are using a HP Tandem 6520 or 6530 terminal emulator, and TACL is configured to clear the screen as a security measure (the default), it does so when you log off. You can override the automatic screen clearing with the NOCLEAR option. Conversely, if TACL is configured to omit automatic screen clearing, you can use the CLEAR option to clear the terminal screen.

#LOOKUPPROCESS Built-In Function

Use #LOOKUPPROCESS to request information about a named process or process pair from the destination control table (DCT).

```
#LOOKUPPROCESS / option [ , option ] ... / specifier
```

option

is a request option. It can be any of these:

ANCESTOR

returns the process name, or the CPU number and process identification number, of the process creator. If the ancestor node is different from the specified or default node, then TACL displays the node name.

BACKUP

returns the CPU,PIN of the backup process. If there is no backup, TACL returns 0,0.

ENTRY

returns the number of the DCT entry whose information is being returned. This option forces specifier to be an entry number.

PRIMARY

returns the CPU,PIN of the primary process in the DCT.

PROCESSID

returns the process name of the DCT entry.

RESULT

returns one of these values:

0 if the requested entry is obtained

1 if an entry number is given but a higher-numbered entry is being returned because the specified entry is not in use

2 if the entry is not in the table or the system cannot be reached

specifier

is one of these process identifiers:

```
[\node-name.]{$process-name | cpu,pin }
entry-number [ \node-name ]
```

node-name

is the system where the process is running.

$*process-name*

is the name of the process.

cpu,pin

is the CPU,PIN of the entry.

entry-number [*node-name*]

is an entry number in the DCT for the specified system. If you omit*node-name*, the current default is used.

Result

#LOOKUPPROCESS returns a space-separated list of the selected information about the DCT entry. The information is listed in the order in which the request options appear.

Considerations

- If a process has no name, it is not listed in the DCT, so #LOOKUPPROCESS returns no information. If you specified RESULT, TACL returns 2.
- If you use the ENTRY option, use an entry number as the DCT specifier.
- If specifier is an entry number, TACL searches the DCT, starting with that entry number, until it finds an entry that is actually in use. For example, if the first entry number in use is 127, the call

  ```
  #LOOKUPPROCESS /ENTRY/ 1
  ```

  returns

  ```
  127
  ```

  Given the same circumstances, the call

  ```
  #LOOKUPPROCESS /PROCESSID, RESULT/ 1
  ```

  returns

  ```
  $Z001 1
  ```

  showing the process name of the first process on the list and indicating that it has a higher entry number than the one specified.

#LOOP Built-In Function

Use #LOOP to execute one or more statements iteratively within a function.

```
#LOOP enclosure
```

enclosure

is an enclosure that can contain either of two sets of two labels: WHILE and DO, or DO and UNTIL.

```
| WHILE | numeric-expression
```

or

```
| UNTIL | numeric-expression
```

evaluates *numeric-expressio*n as either true or false:

```
true     = a nonzero value
false    = zero
```

```
| DO | [ text ]
```

returns text, typically a sequence of one or more functions to be repeatedly executed while *numeric-expression* is true, or until *numeric-expression* becomes true. If you omit text, #LOOP merely waits until the specified criterion is met.

Result

#LOOP evaluates the WHILE or UNTIL text and tests it for inequality to zero.

Depending on the result of that test, #LOOP either returns the DO text (typically TACL statements for execution) and repeats the test, or terminates.

Considerations

- The WHILE/DO version of #LOOP evaluates the expression first; if the expression is true, it executes the DO text. It continues to do so as long as the expression is true. The |WHILE| label must precede the |DO| label.
- The DO/UNTIL version of #LOOP executes the DO text before evaluating the expression; if the expression is false, #LOOP performs another iteration. #LOOP continues its iterations until the expression becomes true. This version always makes at least one loop, even if the expression was initially true. The |DO| label must precede the |UNTIL| label.

#MATCH Built-In Function

Use #MATCH to determine whether a particular text string satisfies the rules for a template.

```
#MATCH template [ text ]
```

template

is a character sequence that may include these template characters:

* matches zero or more characters

? matches a single character

text

is a character sequence to be compared to template.

Result

#MATCH returns -1 if the text satisfies the rules for the template; otherwise, it returns 0.

Considerations

- The comparison is not case-sensitive; that is, an uppercase character is equal to its lowercase counterpart.
- *template* cannot contain spaces; *text* can contain spaces, but TACL construes text with spaces as a space-separated list of multiple arguments. TACL examines only the first argument and ignores the rest. For example, if the variable LINE contains a line of text, you can use:

  ```
  [#IF [#MATCH BEGIN [line]] |THEN| ... ]
  ```

  to determine whether the first word in the line is “begin” (uppercase or lowercase is not a factor).
- To compare one string with another string, use the #COMPAREV built-in function. (Strings can contain embedded spaces.).

Note. Use of a large number of wild-card characters (*,?) can use significant processor resources.

#MOM Built-In Function

Use #MOM to obtain the process name, or CPU number and process identification number, of the creator, or backup, of your TACL process. #MOM invokes the MOM operating system procedure.

```
#MOM
```

Result

- If TACL is a named process and the backup does not exist, the result is blank.
- If TACL is a named process and the backup does exist, the result is the process name itself.
- If TACL is an unnamed single process, the result is the TACL ancestor.

Consideration

To obtain the identity of the job ancestor of your TACL process, if your TACL process is part of a batch job, use the #MYGMOM built-in function.

#MORE Built-In Function

Use #MORE in a routine to find out whether an entire argument set has been processed.

```
#MORE
```

Result

#MORE returns -1 if there are more arguments, 0 if there are none remaining.

Considerations

- To parse the arguments passed to a routine, use the #ARGUMENT built-in function.
- To examine the unprocessed part of an argument set, use the #REST built-in function.
- To obtain the number of characters that #ARGUMENT has processed, use the #GETSCAN built-in function.
- To specify the position for the next #ARGUMENT command to resume processing, use the #SETSCAN built-in function.

#MYGMOM Built-In Function

Use #MYGMOM to obtain the identity of the job ancestor of your TACL process, if your TACL process is part of a batch job.

```
#MYGMOM
```

Result

#MYGMOM returns the process name, or the CPU number and process identification number, of the TACL process job ancestor, if there is one; otherwise, it returns nothing.

Consideration

To obtain the process name, or CPU number and process identification number, of the creator or backup process associated with your TACL process, use the #MOM built-in function.

#MYPID Built-In Function

Use #MYPID to obtain your current CPU number and process identification number (PIN). The number is not qualified with a node name. #MYPID invokes the MYPID operating system procedure.

```
#MYPID
```

Result

#MYPID returns the CPU number and PIN of your TACL process. If it is a process pair, #MYPID returns the CPU,PIN of the primary process.

#MYSYSTEM Built-In Function

Use #MYSYSTEM to obtain the name of the system executing the current TACL. #MYSYSTEM invokes the MYSYSTEMNUMBER operating system procedure and converts the resulting number to the node name.

```
#MYSYSTEM
```

Result

#MYSYSTEM returns the node name.

Consideration

#MYSYSTEM is not affected by #SYSTEM. Although you might be conducting operations on another system, your TACL continues to run on the system on which it was started.

#MYTERM Built-In Variable

Use #MYTERM to obtain the name of your current TACL home terminal. The TACL home terminal is typically the terminal designated by applications to receive critical system messages (such as abend notifications). A home terminal can be a physical terminal name or a process name.

```
#MYTERM
```

Result

#MYTERM returns your current TACL home terminal name.

Considerations

- When you first log on, #MYTERM is initialized to the name of your home terminal.
- Use #MYTERM to direct Inspect and Debug output to your home terminal from a process running at a different terminal.
- Use #MYTERM to change the home terminal name for backup processes.
- When a process abends, TACL notifies the parent TACL process in addition to the home terminal.
- The #MYTERM display includes the node name only if the home terminal is running on a remote system or if the TACL defaults contain a node name.
- Use #PUSH #MYTERM (or PUSH #MYTERM) to save a copy of your current TACL home terminal name.
- Use #POP #MYTERM (or POP #MYTERM) to restore your current TACL home terminal to the last home terminal name pushed.
- Use #SET #MYTERM (or SET VARIABLE #MYTERM) to define your current TACL home terminal name.

  The syntax of #SET #MYTERM is:

  ```
  #SET #MYTERM home-term
  ```

  home-term

  is the terminal name or process name that is to become your new home terminal. This does not affect the location to which TACL writes its output or from which it reads commands, but it does affect the home terminal designation for processes TACL starts.

Examples

1. This example shows how to display the contents of #MYTERM:

```
10> #MYTERM
#MYTERM expanded to:
$TG0.$G04
```

2. This example pushes, sets, and pops the value of #MYTERM:

```
11> #PUSH MYTERM
12> #MYTERM
#MYTERM expanded to:
$TG0.$G04

13> #SET #MYTERM $ZTNT.#PTY46
14> #MYTERM
#MYTERM expanded to:
$ZTNT.#PTY46

15> #POP #MYTERM
16> #MYTERM
#MYTERM expanded to:
$TG0.#G04
```

#NEWPROCESS Built-In Function

Use #NEWPROCESS to start a process (this also establishes the name and CPU,PIN of the default process; see #PROCESS Built-In Function on page 9-290). #NEWPROCESS is similar to the RUN command described in Section 8.

```
#NEWPROCESS program-file [ / option [, option ]... / ]
  [ param-set ]
```

program-file

is the name of the file containing the object program to be run. Partial file names are evaluated using the current default system, volume, and subvolume names.

option

is one of these (see the RUN[D|V] Command on page 8-156 for complete descriptions):

```
CPU cpu-number
DEBUG
DEFMODE { OFF | ON }
EXTSWAP [ file-name ]
HIGHPIN { ON | OFF }
IN [ file-name ]
INLINE
INSPECT { OFF | ON | SAVEABEND }
INV variable-level [ DYNAMIC [PROMPT variable-level] ]
JOBID num
LIB [ file-name ]
MEM num-pages
NAME [ $process-name ]
NOWAIT
OUT [ list-file ]
OUTV variable-level
PFS num-pages
PRI priority
STATUS variable-level
SWAP file-name
TERM $terminal-name
WINDOW [ "text" ]
```

param-set

is a program parameter or a series of parameters sent to the new process in the startup message. Leading and trailing spaces are deleted.

Result

- If you specify NOWAIT, #NEWPROCESS returns the name of the created process, if it is a named process. If it is unnamed, it returns its CPU,PIN.

- If TACL waits for completion of the created process, #NEWPROCESS returns a termination status, a space, and the process identifier of the created process. The termination status is one of these:

STOP	The process ended normally.
ABEND	The process ended abnormally (as the result of an error trap, for example).
CPU	The process was deleted because of a CPU failure.
NET	The process was deleted because of a network failure.

- If TACL is awakened by a WAKE message (code 20) while waiting for completion of the created process, #NEWPROCESS returns WAKE.
- If you press the BREAK key while #NEWPROCESS is waiting, it returns nothing.

Considerations

These conditions apply to the use of the #NEWPROCESS built-in function:

- When a process terminates, the operating system sends TACL a process deletion message that contains completion information. TACL places that information in the variables :_COMPLETION (C-series format) and :_COMPLETION^PROCDEATH (D-series format), if those variables exist. (For additional information about completion codes, see Section 5, Statements and Programs.)
- When running a process that is to communicate with TACL (such as by setting IN or OUT to the TACL process name, or by using TACL variables in INV or OUTV, or by using the INLINE feature), be careful to coordinate TACL functions that enable the communication (such as #IN or #OUT) with the counterpart mechanisms in that process. Deadlock conditions can result if TACL tries to open a process for communication at the same time that process is trying to open TACL for communication.
- In NOWAIT mode, TACL does not wait for the process to finish, but prompts the terminal for the next command immediately. TACL accesses the terminal in conversational mode. Some processes, such as TEDIT, access the terminal in page mode. Two processes cannot share a terminal when one uses conversational mode and the other uses page mode.
- TACL allows IN and OUT files to be DEFINE names, and passes them to the process being executed. The process is responsible for handling the DEFINEs.
- TACL allows TERM names to be DEFINE names, and passes them to the process being executed.
- If you start a TACL process with HIGHPIN OFF, any processes started by that TACL process run at a low PIN by default.
- The #NEWPROCESS built-in function returns the node name only if the process was started on a remote node.

- To obtain additional information about error results, use the #ERRORNUMBERS built-in variable.
- #ERRORNUMBERS returns different values for C-series and D-series #NEWPROCESS calls. For more information, see #ERRORNUMBERS Built-In Variable on page 9-160.
- The #NEWPROCESS built-in function also supports file type 800, the native object code file for TNS/E systems.
- Redirection abilities of the OSH command utility can be used. For more information on redirection, see Section 6, Open System Services Shell and Utilities Reference Manual.

These conditions apply to the use of the RUN command for starting TACL processes:

- A TACL process starts in a logged-off state unless Safeguard software authenticates users and requests that TACL processes start in a logged-on state.
- To run TACL as a server process, set the IN file to $RECEIVE. For more information, see the *TACL Programming Guide*.
- If the IN file is the same as the OUT file and the TACL process is not named, TACL does not set its home terminal.
- The OSH process should not be started with the INLINE option. If you use the INLINE option:
  - TACL will not be able to detect the end of the INLINE process.
  - The output displayed might not be synchronized with the command entered using the inline prefix.

#NEXTFILENAME Built-In Function

Use #NEXTFILENAME to obtain the file whose name follows the specified file alphabetically on a given volume. #NEXTFILENAME invokes the NEXTFILENAME operating system procedure.

```
#NEXTFILENAME [ file-name ]
```

`file-name`

is the file name after which to begin the search for the next file name. Partial file names are expanded using the current defaults.

Result

#NEXTFILENAME returns the name of the located file.

Consideration

If you omit *file-name*, #NEXTFILENAME returns the first file in the current default volume and subvolume.

#OPENINFO Built-In Function

Use #OPENINFO to obtain information about file openers.

```
#OPENINFO / option [ , option ] /
  { file-name | device-name } tag
```

option

is an information request. It can be any of these:

ACCESSID

returns the user number of the opener.

ACCESSMODE

returns the access mode with which the file or device was opened:

0 Read/write

1 Read-only

2 Write-only

BACKUP

returns the CPU,PIN of the backup process, if there is one and that process has
the file open; otherwise, it returns nothing.

EXCLUSION

returns the exclusion specification under which the file or device was opened:

0 Shared

1 Exclusive

3 Protected

FILENAME

returns the name of the file opened (can be useful when the argument is a disk name).

PRIMARY

returns the CPU,PIN of the primary process.

PROCESSID

returns the process name of the opening process, if the process is named; otherwise, it returns the CPU,PIN of that process.

SYNCDEPTH

returns the sync depth with which the file or device was opened.

file-name or *device-name*

identifies the file or device for which information is desired. Each call to #OPENINFO returns information about a single opening of the file or device.

tag

is a number identifying the particular opening for which information is wanted. A zero value obtains information about the first opening; #OPENINFO includes in its result the tag value to be used to get information about the next opening in sequence.

Result

#OPENINFO returns a file-system code indicating the outcome of the operation:

0 Success

1 No (more) openings

Other file errors may be returned. See the *Guardian Procedure Errors and Messages Manual* for the meanings of such messages.

If the result is 0, it is followed by a space, then the tag to be passed in the next call to #OPENINFO, then another space, and finally a space-separated list of the information requested by the options you supplied, in the order requested.

Considerations

- Openings are not reported in any defined order. In particular, when #OPENINFO is retrieving information about all openings of a disk volume, the openings for any one file are not grouped together in the sequence of calls.
- Because the BACKUP option may return nothing, you should specify it as the last option to prevent confusion in reading the information list.

Example

This routine accepts a file or device name and “processes” (simulation only) all the openings of that file or device.

```
?SECTION lookopen ROUTINE
#FRAME

== Make some temps
#PUSH searchname error tag processid primary backup

== Get the argument
#IF [#ARGUMENT/VALUE searchname/ FILENAME DEVICE]
#IF [#ARGUMENT END]

== Start with first opening
#SET tag 0

== Loop until error
[#LOOP |DO|

== Get information using previous tag
[#SETMANY
  error tag processid primary backup
  ,
  [#OPENINFO/PROCESSID,PRIMARY,BACKUP/ [searchname] [tag]]
]
== Check for error
[#IF (error = 0) |THEN|

  == No error; process the open-information
  ...

] { End IF }

|UNTIL| (error <> 0)
] { End LOOP }

== Error 1 is normal termination
[#IF (error <> 1) |THEN|
  #OUTPUT Error [error] on [searchname]
]

== Clean up
#UNFRAME
```

#OUT Built-In Variable

Use #OUT to set or obtain the name of the file currently being used by TACL as its OUT file.

```
#OUT
```

Result

#OUT returns the name of the current TACL OUT file.

Considerations

When you first log on to an interactive TACL, #OUT is initialized to the name of your home terminal.

You cannot permanently change the primary TACL OUT file; that is the primary OUT file always remains the same. You must push #OUT before you set it to a new OUT file.

Setting #OUT to a disk file requires that TACL allocate one of its block buffers internally. Because these blocks are large and must be allocated from the first 64K bytes of the TACL address space, there are only four of them. (Other consumers of these blocks buffers are #IN and #REQUESTER.)

When TACL has been started with its IN set to $RECEIVE, these guidelines apply:

- You can set #OUT without first pushing it. This feature, in conjunction with an equivalent one in #IN, allows you to run TACL as a server yet be able to take over a terminal as though TACL had been run on the terminal in the usual way.
- The value of #OUT is the default OUT file for processes started while #OUT is set in this way.
- You can revert to sending output as a REPLY to $RECEIVE by setting #OUT to a null value.
- To set or obtain the name of the current IN file, use the #IN built-in variable.
- Lines read from the current IN are appropriately echoed to the current OUT. The default OUT file to processes started by TACL is not affected by #OUT; the default remains associated with the original TACL OUT file.
- When communicating with a process, be careful to coordinate functions that enable the communication (such as #IN or #OUT) with the counterpart mechanisms in that process (such as IN or OUT referring to the TACL process name). Deadlock conditions can result if TACL tries to open a process for communication at the same time that process is trying to open TACL for communication.
- Use #PUSH #OUT (or PUSH #OUT) to save a copy of the current OUT file name.

Any error while pushing OUT causes #OUT to be popped at once. Any other error or break occurring while #OUT is pushed causes all but the bottom level of #OUT to be popped before the error message is printed.

- Use #POP #OUT (or POP #OUT) to restore the last OUT file name pushed.
- Use #SET #OUT (or SET VARIABLE #OUT) to set the name of the file to be used by TACL for OUT. Before setting #OUT, you must #PUSH #OUT (unless IN is set to $RECEIVE, as noted previously).

The syntax of #SET #OUT is:

```
#SET #OUT file-name
```

file-name

is the name you give the OUT file. You can use the name of a process in place of a disk file name; TACL writes output to the process as though it were a file.

If you specify a nonexistent disk file, TACL creates an edit-format file. If you specify an existing disk file, lines are appended to the existing file.

Example

This example causes TACL to write 002, 040, and 120 to the file OUTFILE, and then the OUT file reverts to its previous status:

```
#PUSH #OUT
...
#SET #OUT outfile
#OUTPUT /HOLD,COLUMN 5,JUSTIFY RIGHT,WIDTH 3,FILL ZERO/ 2
#OUTPUT /HOLD,COLUMN 13,JUSTIFY RIGHT,WIDTH 3,FILL ZERO/ 40
#OUTPUT /COLUMN 21, JUSTIFY RIGHT, WIDTH 3, FILL ZERO/ 120
#POP #OUT
```

#OUTFORMAT Built-In Variable

Use #OUTFORMAT to set or obtain the current formatting mode for the TACL OUT file.

```
#OUTFORMAT
```

Result

#OUTFORMAT returns the current #OUTPUT formatting mode: PLAIN, PRETTY, or TACL.

Considerations

- When you first log on, #OUTFORMAT is initialized to PLAIN.
- To set or obtain the current formatting mode for the TACL IN file, use the #INFORMAT built-in variable.
- When #OUTFORMAT is set to TACL, metacharacters that are stored as plain characters on input (such as {, }, ==, [,], and |) are preceded with a tilde on output.
- Use #PUSH #OUTFORMAT (or PUSH #OUTFORMAT) to save a copy of the current #OUTPUT formatting mode.
- Use #POP #OUTFORMAT (or POP #OUTFORMAT) to restore the last #OUTPUT formatting mode pushed.
- Use #SET #OUTFORMAT (or SET VARIABLE #OUTFORMAT) to set the way special internal notations are printed by #OUTPUT and by the prompt part of #INPUT.

The syntax of #SET #OUTFORMAT is:

```
#SET #OUTFORMAT { PLAIN | PRETTY | TACL }
```

PLAIN

specifies that TACL should not translate any characters written to the OUT file. Because internal representations include nonprinting characters, displaying TACL program data in PLAIN mode can produce illegible output, depending on how your device interprets nonprinting characters.

PRETTY

causes TACL to display the internal representations of square brackets and vertical lines, as well as the actual characters, as brackets and vertical lines. The tilde and underscore combination (~_) produces an ordinary space. You can use this feature to manage output spacing or to include a trailing space in a prompt, as follows:

```
#FRAME
#PUSH #OUTFORMAT
```

```
#SET #OUTFORMAT PRETTY
#INPUT Prompt:~_
...
#UNFRAME

TACL
```

causes TACL to display the internal representations of square brackets ([and]) and the vertical line (|) as those characters. It shows actual metacharacters preceded by tildes (~).

#OUTPUT Built-In Function

Use #OUTPUT to write data to the TACL OUT file.

```
#OUTPUT [ / option [ , option ] ... / ] [ text ]
```

option

qualifies the write operation. It can be any of these:

`COLUMN` *num*

begins writing data at column num. If there is already text in the buffer extending beyond the specified column, the buffer contents are output and a new line is started, beginning at that column.

`FILL { SPACE | ZERO }`

specifies the character to be used to fill the unused character positions to the right if the output is narrower than the number of columns specified by WIDTH. If the output is wider than the number of columns specified by WIDTH, FILL is ignored.

`HOLD`

holds output in a buffer until one of these events occurs:

- #OUTPUT or #OUTPUTV is executed without the HOLD option.
- The buffer becomes full.
- TACL, #DELTA, #INPUT, or #INPUTV prompts for input.
- #DELTA exits.

`JUSTIFY { LEFT | RIGHT | CENTER }`

specifies whether the output should be left-justified, right-justified, or centered if the output is narrower than the number of columns specified by WIDTH. If the output is wider than the number of columns specified by WIDTH, JUSTIFY is ignored.

`WIDTH` *num*

specifies the width of the output field for the FILL and JUSTIFY options. If the output is wider than the number of columns specified by WIDTH, WIDTH is ignored. Specify num as an integer in the range from 1 to the current value of the #WIDTH built-in variable.

WORDS

causes text to be treated as a space-separated list of output items, applying the FILL, JUSTIFY, and WIDTH options to each item. Without WORDS, text is treated as a single output item.

text

is the output item. If you omit text, #OUTPUT writes a blank line.

Result

#OUTPUT returns nothing.

Considerations

- This statement outputs the first line of vara and then executes any other lines, unless vara is a routine:

```
#OUTPUT [vara]
```

If vara is a routine, TACL interprets the contents of vara and outputs its results.

- Enclose the variable name in square brackets ([#OUTPUT [vara]]) or use #OUTPUTV (or OUTVAR) vara to output the contents of a multiple-line variable.
- If text begins with a slash (/), you must supply an option to #OUTPUT so that the slash at the beginning of text is not interpreted as the start of a list of options.
- #OUTPUT accepts a text argument, which does not include end of line characters.
- When you use the HOLD option, you can construct output lines in pieces. This code displays “Alltogethernow”:

```
?TACL MACRO
#OUTPUT /HOLD/ All
#OUTPUT /HOLD/ together
#OUTPUT now
```

If, however, you want your output on separate lines, you can force separate lines by specifying the COLUMN option; TACL starts a new line if there is output in the specified column. This macro:

```
?TACL MACRO
#OUTPUT /COLUMN 5, HOLD/ This is the first line
#OUTPUT /COLUMN 5, HOLD/ This is the second line
#OUTPUT /COLUMN 5, HOLD/ This is the third line
#OUTPUT
#OUTPUT This is the last line
```

displays the following:

```
5> test
     This is the first line
     This is the second line
```

```
     This is the third line
This is the last line
```

The fourth #OUTPUT call leaves room for the third line; otherwise, it would appear on the same line as the last line.

#OUTPUTV Built-In Function

Use #OUTPUTV to write the contents of a string to the OUT file of TACL (typically, the terminal).

```
#OUTPUTV [ / option [ , option ] ... / ] string
```

option

qualifies the write operation. It can be any of the following unless string is a variable level of type STRUCT, in which case no options are allowed.

`COLUMN` *num*

begins writing data at column num. If there is already text in the buffer extending beyond the specified column, the buffer contents are output and a new line is started, beginning at that column.

`FILL { SPACE | ZERO }`

specifies the character to be used to fill the unused character positions to the right if the output is narrower than the number of columns specified by WIDTH. If the output is wider than the number of columns specified by WIDTH, FILL is ignored.

`HOLD`

holds output in a buffer until one of these events occurs:

- #OUTPUT or #OUTPUTV is executed without the HOLD option.
- The buffer becomes full.
- TACL, #DELTA, #INPUT, or #INPUTV prompts for input.
- #DELTA exits.

`JUSTIFY { LEFT | RIGHT | CENTER }`

specifies whether the output should be left-justified, right-justified, or centered if the output is narrower than the number of columns specified by WIDTH. If the output is wider than the number of columns specified by WIDTH, JUSTIFY is ignored.

`WIDTH` *num*

specifies the width of the output field for the FILL and JUSTIFY options. If the output is wider than the number of columns specified by WIDTH, WIDTH is ignored. Specify num as an integer in the range from 1 to the current value of the #WIDTH built-in variable.

WORDS

specifies that each line of string is to be treated as a space-separated list and that the FILL, JUSTIFY, and WIDTH options are to be applied to the individual members of the list. Without WORDS, string is treated as a single output item.

string

is the data to be output. It is the name of an existing variable level, text enclosed in quotation marks, or a concatenation of such entities. The concatenation operator is '+' (the apostrophes are required).

Result

#OUTPUTV returns nothing.

Considerations

- If you supply options, they are applied to each line of string as though you had called #OUTPUT with the same options once for each line. In particular, the last line is the only one that can be held, as each line forces output of any previous line.
- #OUTPUTV with the /HOLD/ option is useful for constructing lines to be output piece by piece. For example, this example illustrates how you can do this by putting the commands into an edit-format file and then invoking the file.

```
?TACL MACRO
#PUSH vara
#SET vara What a
#OUTPUTV /HOLD/ vara
#SET vara fine
#OUTPUTV /HOLD/ vara
#SET vara day
#OUTPUTV vara
```

When you invoke the macro, the text all comes out on one line.

- No options are allowed when a STRUCT is being output.
- Redefinitions are not shown unless string is itself a redefinition.
- You can use #OUTPUTV to display the current values of a structure, substructure, or structure item in a stylized format defined by TACL. For example:

```
23> #OUTPUTV inventory
item(0:0) 123
price(0:0) 1004
quantity(0:0) 97
```

Conversely, #OUTPUT [inventory], displays only a space-separated list of the structure values.

Example

This example shows the use of a concatenated string containing both a variable name and quoted text as an argument to #OUTPUTV.

```
#PUSH termname
#SET termname [#MYTERM]
#OUTPUTV "My terminal is " '+' termname '+' " at this time."
```

Assuming the home terminal name is $GREEN at the time the #SET function is executed, the following is written to the OUT file:

```
My terminal is $GREEN at this time.
```

#PARAM Built-In Variable

Use #PARAM to set or obtain the value of a specified parameter or to obtain a list of all the parameters you have currently in effect. For a discussion about the use of PARAMs, see the *TACL Programming Guide*.

```
#PARAM [ param-name ]
```

param-name

is the name of the parameter whose value is to be determined.

Result

#PARAM returns the value of the specified parameter or, if you omit *param-name*, it returns a space-separated list of all your parameters.

Considerations

- When you first log on, #PARAM is initialized to a null value.
- TACL reserves 1024 bytes of internal storage for parameters and their values. The number and length of parameters in effect are limited by this storage area.
- When a backup TACL process takes over, TACL clears all parameters.
- Use #PUSH #PARAM (or PUSH #PARAM) to save a copy of all your parameters.
- Use #POP #PARAM (or POP #PARAM) to replace all your current parameters with those last pushed.
- Use #SET #PARAM (or SET VARIABLE #PARAM) to define a parameter name and, optionally, assign a value to the parameter. These considerations apply:
  - If you supply both a parameter name and value, the specified parameter is set.
  - If you omit both the name and the value, all current parameters are cleared.
  - If you supply only a parameter name, the parameter is deleted.

  The syntax of #SET #PARAM is:

  ```
  #SET #PARAM [ param-name [ param-value ] ]
  ```

  param-name

  is the name of the parameter to be assigned a value. It can contain from 1 to 31 alphanumeric characters, including circumflex (^) and hyphen (-).

param-value

is the value assigned to *param-name*. It has a maximum length of 255 characters and must be no longer than one logical line. Specify *param-value* as either character-sequence or "character-sequence".

If you do not use quotation marks, you cannot include any commas or spaces as part of the parameter value.

If you do use quotation marks, all characters between the quotation marks are included as part of the parameter value. To include quotation marks in the parameter value, enter them twice (““). (The enclosing quotes, and one of each pair of doubled quotes, do not count against the 255-character size limit.)

Use two adjacent quotes to express an empty *param-value*.

#PAUSE Built-In Function

Use #PAUSE to give control of your terminal to processes other than the current TACL.

```
#PAUSE [ [\node-name.]{$process-name | cpu,pin } ]
```

\node-name

is the name of the system on which the process resides.

$*process-name*

is the name of the process or process pair for whose termination TACL is to wait.

cpu,pin

is the CPU number and process identification number of the process.

Result

#PAUSE returns one of these:

- Upon completion of the process, #PAUSE returns a termination status, a space, and the process name, or the CPU number and the process identification number, of the process. The termination status is one of these: STOP, ABEND, CPU, or NET (see #NEWPROCESS Built-In Function on page 9-265).
- If TACL is awakened while waiting for the completion of a process, #PAUSE returns WAKE.
- If TACL is awakened by the BREAK key, #PAUSE returns nothing.

Considerations

- #PAUSE does not specify which process can access your terminal.
- When you enter #PAUSE, the current TACL stops prompting for commands, thus allowing any other processes to gain control of your terminal. TACL regains control of your terminal, again displaying its prompt (n>), when it receives a process deletion message from the specified process.
- If the process you specify (*process-name* or *cpu,pin*) does not exist, or was not created by the current TACL, #PAUSE waits until you press the BREAK key or TACL receives a WAKE message.
- If you do not specify a process, #PAUSE waits for the last process created by the current TACL. If that process has already terminated, #PAUSE waits until you press the BREAK key or TACL receives a WAKE message.
- A process specification in #PAUSE establishes the identity of the default process; see the #PROCESS Built-In Function on page 9-290 for information on the default process.

#PMSEARCHLIST Built-In Variable

Use #PMSEARCHLIST to set or obtain the current contents of the search list used for finding program and macro files.

```
#PMSEARCHLIST
```

Result

#PMSEARCHLIST returns the current contents of the search list used for finding programs and macro files.

Considerations

- #PMSEARCHLIST has a maximum size of 500 characters.
- When you first log on, #PMSEARCHLIST is initialized to $SYSTEM.SYSTEM.
- TACL uses the #PMSEARCHLIST variable for remote program files as well as local program files. To specify a separate search list for remote files, #PUSH and #SET #PMSEARCHLIST prior to starting the remote process. When finished, #POP #PMSEARCHLIST.
- Use #PUSH #PMSEARCHLIST (or PUSH #PMSEARCHLIST) to save a copy of the current contents of the program and macro search list.
- Use #SET #PMSEARCHLIST (or SET VARIABLE #PMSEARCHLIST) to define a search list to be used by TACL when searching for a program or macro file. A search list is a space-separated list of subvolumes.

  The syntax of #SET #PMSEARCHLIST is:

  ```
  #SET #PMSEARCHLIST searchlist
  ```

  searchlist

  is a space-separated list of subvolumes to be searched when you give TACL the name of a program or macro file.

  When the search list is used, each of its entries is defaulted with the current default subvolume and the resulting subvolume is searched for the file. When the file is found, further subvolumes are not used.

  - You can #SET #PMSEARCHLIST #DEFAULTS (no brackets around #DEFAULTS), which will later use whatever defaults are in effect at the time you use the search list; for example:

    ```
    35> VOLUME $guest.alice
    36> #SET #PMSEARCHLIST #DEFAULTS
    37> VOLUME $new.jones
    38> test
    ```

In this case, TACL searches for $NEW.JONES.TEST.

- You can #SET #PMSEARCHLIST [#DEFAULTS] (with brackets around #DEFAULTS), which uses whatever your current subvolume is at the time you set the search list; for example:

```
39> VOLUME $guest.alice
40> #SET #PMSEARCHLIST [#DEFAULTS]
41> VOLUME $new.jones
42> test
```

In this case, TACL searches for $GUEST.ALICE.TEST

- Use #POP #PMSEARCHLIST (or POP #PMSEARCHLIST) to restore the program and macro search list from the last copy pushed.

Note. Only subvolume names are allowed in #PMSEARCHLIST.

#PMSG Built-In Variable

Use #PMSG to set or obtain the current state of the PMSG flag. If the PMSG flag is on, TACL outputs a message giving the process name, the CPU number, and process identification number of each process you start, and another message each time a process you started stops.

```
#PMSG
```

Result

#PMSG returns the current state of the PMSG flag: 0 if it is off, -1 if it is on.

Considerations

- When you first log on, #PMSG is initialized to zero.
- Use #PUSH #PMSG (or PUSH #PMSG) to save a copy of the current setting of the PMSG flag.
- Use #POP #PMSG (or POP #PMSG) to restore the PMSG flag from the copy last pushed.
- Use #SET #PMSG (or SET VARIABLE #PMSG) to set the PMSG flag on or off.

  The syntax of #SET #PMSG is:

  ```
  #SET #PMSG num
  ```

  num

  is a nonzero value to turn the flag on, zero to turn it off.

#POP Built-In Function

Use #POP to delete the top level of one or more variables.

```
#POP variable [ [,] variable ] ...
```

variable

is the name of an existing variable or the name of a built-in variable.

Result

#POP returns nothing.

Considerations

- If the top level is the only level, #POP deletes the variable (except for built-in variables.
- When it encounters an #UNFRAME, TACL performs an implicit #POP for every #PUSH (or PUSH) that was done since the most recent #FRAME was issued.
- TACL performs an implicit #POP on #IN when #INPUT /UNTIL EOF/ or #INPUTV /UNTIL EOF/ is used.
- An attempt to pop a variable that has not been pushed produces an “Expecting an existing variable” message; an attempt to pop a built-in variable that has not been pushed produces a “Was not pushed” message.
- Do not try to #POP the root directory (:). If you try this, TACL returns "*ERROR* Cannot push or pop the root segment's root." In addition, to avoid losing standard functionality from your TACL environment, do not pop the directories supplied as part of the TACL software product (such as UTILS).

#PREFIX Built-In Variable

Use #PREFIX to set or obtain the current contents of the prompt prefix string.

```
#PREFIX
```

Result

#PREFIX returns the current contents of the prompt prefix string.

Considerations

- When you first log on, #PREFIX is initialized to a null value.
- Use #PUSH #PREFIX (or PUSH #PREFIX) to save a copy of the current contents of the prefix string.
- Use #POP #PREFIX (or POP #PREFIX) to restore the prefix string from the last copy pushed.
- Use #SET #PREFIX (or SET VARIABLE #PREFIX) to define text to be output by TACL at the beginning of any prompt (except for those output by #INPUT) if #PROMPT is not zero. (See #PROMPT Built-In Variable on page 9-312 for a way to recompute #PREFIX each time it is used as a prompt.)

  The syntax of #SET #PREFIX is:

  ```
  #SET #PREFIX [ text ]
  ```

  text

  is a prefix of up to 40 characters. If you omit text, there is no prefix.

Example

1. This example illustrates the use of #PREFIX to output the current prompt prefix:

   ```
   MINE 20> #OUTPUT [#PREFIX]
   MINE
   MINE 21>
   ```

2. This example shows the relationship between _PROMPTER, #PREFIX, and #PROMPT:

   ```
   63> [#DEF _prompter MACRO |BODY| == Define a macro
   63> #SET #PREFIX Number
   63> ]
   64> #SET #PROMPT -1 == Enable the custom prompt
   Number 65>
   ```

#PROCESS Built-In Function

Use #PROCESS to obtain the process name, or the CPU number and process identification number of the last process created by TACL or for which TACL last paused. This is the default process used by #ABEND, #ACTIVATEPROCESS, #ALTERPRIORITY, #PAUSE, #STOP, and #SUSPENDPROCESS.

```
#PROCESS
```

Result

#PROCESS returns the name of the default process or, if it has no name, its CPU,PIN; if the process has been deleted, #PROCESS returns nothing.

Consideration

The #PROCESS built-in function returns a node name only if the process is running on a remote node or if a remote node name is specified in the current defaults.

#PROCESSEXISTS Built-In Function

Use #PROCESSEXISTS to find out whether a given process exists.

```
#PROCESSEXISTS [\node-name.]{$process-name | cpu,pin }
```

\node-name

is the name of the system on which the process resides.

$process-name

is the name of the process or process pair.

cpu,pin

is the CPU number and process identification number of the process.

Result

#PROCESSEXISTS returns -1 (true) if the process exists; otherwise, it returns 0 (false).

Considerations

To obtain information about a process, use the #PROCESSINFO built-in function.

#PROCESSFILESECURITY Built-In Variable

Use #PROCESSFILESECURITY to obtain the current file-creation security for the current TACL process.

```
#PROCESSFILESECURITY
```

Result

#PROCESSFILESECURITY returns the current file-creation security, enclosed in quotes, for this TACL process.

Considerations

- When you first log on, #PROCESSFILESECURITY is initialized to your saved default security.
- Use #PUSH #PROCESSFILESECURITY (or PUSH #PROCESSFILESECURITY) to save a copy of the current setting of your TACL file-creation security.
- Use #POP #PROCESSFILESECURITY (or POP #PROCESSFILESECURITY) to restore your TACL file-creation security from the last copy pushed.
- Use #SET #PROCESSFILESECURITY (or SET VARIABLE #PROCESSFILESECURITY) to set the four-character security string that specifies the security to be given to each file created by this TACL.

  The syntax of #SET #PROCESSFILESECURITY is:

  ```
  #SET #PROCESSFILESECURITY "security"
  ```

  "*security*"

  is a four-character security string enclosed in quotation marks. The four characters represent the four security attributes:

  ```
  R W E P
  ```

R	Specifies who can read the file
W	Specifies who can write to the file
E	Specifies who can execute the file
P	Specifies who can purge the file

- Each security attribute (R, W, E, or P) can be any of these characters:

  O (Owner) Only the owner can access the file; the owner must be logged on to the local system.

  G (Group) Anyone in the owner's group can access the file; the user must be logged on to the local system.

  A (Anyone) Any user can access the file; the user must be logged on to the local system.

  U (User) Only the owner can access the file; the owner can be logged on to the local system or a remote system.

  C (Community) Anyone in the owner's group can access the file; the user can be logged on to the local system or a remote system.

  N (Network) Any user can access the file; the user can be logged on to the local system or a remote system.

  - Only the local super ID can access the file.

- When you log on, your security is set to the configuration you last specified with the DEFAULTS utility.

#PROCESSINFO Built-In Function

Use #PROCESSINFO to request information about a process.

```
#PROCESSINFO / option [ , { option | search-option } ] ... /
  [ [ \node-name.] { $process-name | cpu,pin } ]
```

option

is a request option. It can be any of these:

CAID

returns the creator accessor ID as (*group-num,user-num*).

CONTEXTCHANGES

is the number of changes made to the DEFINE process context since process creation. The number returned is modulo 65536.

CPU

returns the central processing unit number.

CURRENTMAINSTACKSIZE

returns the current main stack size in bytes. Nothing is returned if the target system is running an operating system RVU earlier than D40 or the main stack is currently not used. An error message can be returned, indicating the option is invalid.

CURRENTNATIVEHEAPSIZE

returns the current size of the native heap area in bytes. Nothing is returned if the target system is running an operating system RVU earlier than D40 or the native heap area is currently not used. An error message can be returned, indicating the option is invalid.

DEFINEMODE

equals 1 if DEFINEs are enabled; otherwise, it equals 0.

EXTSWAP

is the name of the current extended swap file. Nothing is returned if the process has no extended swap file.

GMOMJOBID

returns the job-ancestor.job-id of the process, if it belongs to a batch job; otherwise, it returns nothing.

GUARANTEEDSWAPSPACE

returns the amount of swap space reserved for use by the process in bytes. It returns nothing if the target system is running an operating system RVU earlier than D40. An error message can be returned, indicating the option is invalid.

HOMETERM

returns the name of the home terminal.

INITPRI

is the initial priority of a process when it begins execution or as changed by a call to the PRIORITY or PROCESS_SETINFO_ operating system procedures or related TACL commands and built-in functions such as ALTPRI, #ALTERPRIORITY, and #PROCESS. (The current priority might be less because of the sliding priority algorithm used by the operating system.)

IPUASSOCIATION

returns the IPU affinity attributes for the process indicating whether or not the process is bound to an IPU and whether or not it is bindable. This attribute is undefined if the system is running an operating system version earlier than H06.27 or J06.16.

IPUNUMBER

returns the number of the IPU on which the process last ran.

LOGONNAME

returns the user name (*group name,user name*), user ID (*group number,user number*), or user alias specified in the current LOGON command. For pre-D30 software RVUs, this option returns nothing.

LIBRARY

returns the name of the library file used by the process. If the process has no associated library file, this option returns nothing.

LICENSES

equals 1 if the object file was licensed at process creation time; otherwise, it equals 0.

LOGONSTATE

equals 1 if the process is logged on; otherwise, it equals 0.

MAXMAINSTACKSIZE

returns the maximum size, in bytes, to which the main stack can grow. It returns nothing if the target system is running an operating system RVU earlier than D40. An error message can be returned, indicating the option is invalid.

MAXNATIVEHEAPSIZE

returns the maximum size, in bytes, to which the native heap area can grow. It returns nothing if the target system is running an operating system RVU earlier than D40. An error message can be returned, indicating the option is invalid.

MOM

is the process identifier of the MOM of the process. If the process is named, #PROCESSINFO returns the name; otherwise, CPU and PIN are returned. The MOM process is always qualified with a node name. If the process has no existing parent, nothing is returned.

- When a process is part of a process pair, the MOM of the process is the other member of the pair.
- When a process is unnamed, the MOM of the process is usually the process that created it.
- For a named single process, nothing is returned.

More information about the MOM process can be found in the *Guardian Procedure Calls Reference Manual* and the *Guardian Programmer's Guide*.

NATIVE

returns 1 if the process is a "native" process (that is, runs on a RISC-based operating system without interpretation or translation), returns 0 if the process is a "non-native" process. For pre-D40 software RVUs, this option returns nothing.

OSSARGUMENTS

returns the first 80 bytes of the arguments of the command that created the OSS process. The arguments that are returned may or may not be separated by spaces. If the process is not an OSS process, this option returns nothing. For pre-D30 software RVUs, this option returns nothing.

OSSPATHNAME

returns a fully qualified OSS program pathname if the process is an OSS process. The maximum length of an OSS program pathname (in the OSS file name format) is 1024 bytes. If the process is not an OSS process or if the OSS program is running on a remote node, this option returns nothing. For pre-D30 software RVUs, this option returns nothing.

OSSPID

returns an OSS process ID if the process is an OSS process. An OSS process ID is an integer value. If the process is not an OSS process, this option returns nothing. For software RVUs preceding the D30 RVU, this option returns nothing.

PAID

returns the process accessor ID as (*group-num,user-num*).

PFR

(an abbreviation for privileged-fault-ready) returns a three-digit number. If there is a 1 in the leftmost digit position, it indicates that the process is privileged; a 1 in the second position indicates that the process is waiting because of a page fault; a 1 in the third position indicates that the process is on the ready list. The digits are zero otherwise.

PIN

returns the process identification number.

PRI

returns the priority of the process.

PRIMARY

equals 1 if the process is the primary of a named process pair; otherwise, it equals 0.

PROCESSCREATIONTIME

returns the Julian timestamp that identifies the time when the process was created. This attribute is supported on H and J series systems.

PROCESSDESCRIPTOR

is a process descriptor that includes the node name and sequence number.

PROCESSFILESECURITY

is the default file security for files created by the specified process. Four uppercase characters are returned, enclosed in double quotes.

PROCESSID

returns the process name. If the process has no name, PROCESSID returns the CPU and PIN of the process.

PROCESSSTATE

returns a line of twelve space-separated values: eleven flags and a numeric value. The flags are encoded as -1 (true) or 0 (false), and represent this information:

Flag	Meaning
1	Privileged process
2	Page fault occurred
3	Process is on the ready list
4	System process
5	Reserved
6	Reserved
7	Memory access breakpoint (MAB) in system code
8	Process not accepting any messages
9	Temporary system process
10	Process has logged on (called VERIFYUSER)
11	In a pending process state

The numeric value indicates the state of the process, as follows:

Value	State
0	Unallocated
1	Starting
2	Runnable
3	Suspended
4	DEBUG MAB
5	DEBUG breakpoint
6	DEBUG trap
7	DEBUG request
8	INSPECT MAB
9	INSPECT breakpoint
10	INSPECT trap
11	INSPECT request
12	SAVEABEND
13	Terminating
14	XIO initialization (not applicable on G-series RVUs)

PROCESSTIME

returns the central processing unit time in microseconds that the process has consumed (as opposed to elapsed run time). If the process is remote and is running on a NonStop 1+ system, it returns 0.

PROCESSTYPE

returns an unsigned integer value that indicates the process type. If the process is a Guardian process it returns 0. If the process is an OSS process, it returns 1.

PROGRAMDATAMODEL

returns 1 if the target process is a 64-bit process; otherwise 0 is returned. 0 is returned if the target system is running an operating system version earlier than H06.24 or J06.13.

PROGRAMFILE

returns the program file name. If the process is a system process, is remote, and is running on a NonStop 1+ system, it returns nothing.

If the process is an OSS process, it returns the generic Guardian ZYQ-file name.

If the program file name for a process cannot be retrieved, "`*ERROR* Path to program file name down`" is written to the output file, and the TACL program stops. This situation can occur if the processor for the disk process controlling the disk where the program file resides fails, making the process file name unavailable.

QUALINFOAVAIL

equals 1 if the process has called the PROCESS_SETINFO_ operating system procedure to declare that it supports qualifier name searches through the FILENAME_FIND_ operating system procedures; otherwise, it equals 0.

REMOTECREATOR

equals 1 if the process's creator is remote; otherwise, it equals 0.

RESULT

returns an indication of the success of the #PROCESSINFO call, as follows:

Value	Meaning
0	Results refer to the process specified.
1	Results refer to a higher-numbered process, stepping successively up through PINs and then through CPUs.

Value	Meaning
2	No such process could be found.
5	The system could not be accessed.
99	The parameters were inconsistent.

SRLFILES

returns a list of shared run-time library (SRL) file names used by the process. A maximum of 32 SRLs can be attached to a process. If the process has no associated SRLs, this option returns nothing. If the operating system RVU precedes the D40 RVU, this option returns nothing.

SRLNAMES

returns a list of SRL names used by the process. A maximum of 32 SRLs can be attached to a process. If the process has no associated SRLs, this option returns nothing. If the operating system RVU precedes the D40 RVU, this option returns nothing.

SRLNUMFILES

returns the number of SRL files used by the process. A maximum of 32 SRLs can be attached to a process. If the process has no associated SRLs, this option returns 0. This value is the number of SRL files returned from the system procedure PROCESS_GETINFOLIST_. Some of the SRL file entries can be empty. If the operating system RVU precedes the D40 RVU, this option returns nothing.

SRLNUMNAMES

returns the number of SRL names used by the process. A maximum of 32 SRLs can be attached to a process. If the process has no associated SRLs, this option returns 0. This value is the number of SRL files returned from the system procedure PROCESS_GETINFOLIST_. Some of the SRL file entries can be empty. If the operating system RVU precedes the D40 RVU, this option returns nothing.

SUBDEVICE

is a subdevice type.

SWAP

returns the name of the file used for storing the virtual data of a process. This name is specified in the SWAP *swap-file* option of the RUN command or with the SET SWAP [$*volume-name*] command. If nothing is specified with either (or both) the RUN command SWAP option or SET SWAP command, a dummy file name, "$*volume*.#0", is returned. In this case, #*volume* is the name of the physical volume that the operating system has

selected for storing the file. For pre-D40 software RVUs, this option returns nothing. For more information on the swap facility, see the *Kernel-Managed Swap Facility (KMSF) Manual*.

SYSTEM

returns the name of the system.

UPB

returns a single character (U, P, or B) that indicates whether the process is unnamed or, if named, is the primary or backup of the process pair.

WAITSTATE

returns a space-separated list of eight flags encoded as -1 (true) and 0 (false), as follows:

Flag	Process is Waiting for:
1	PON (CPU power on)
2	IOPON (I/O power on)
3	INTR (interrupt)
4	LINSP (INSPECT event)
5	LCAN (message system, cancel)
6	LDONE (message system, done)
7	LTMF (TMF request)
8	LREQ (message system, request)

WT

returns a three-digit octal value, as follows:

Value	Meaning
%000	Process is running; or process was waiting for an event that has since occurred, and is now ready to run; or process is in a call to DELAY; or process is suspended.
%001	Process is waiting for a message to appear in its $RECEIVE file.
%002	Process is waiting for a TMF request to be completed; or user process is waiting for ENDTRANSACTION to be completed.
%004	Process is waiting for an I/O or interprocess request to be completed.
%005	Process is waiting for a call to AWAITIO for I/O completion on any file.

search-option

is a search option for use with one or more of the preceding options, as follows:

SEARCH *criterion*

specifies a criterion that the process must match; if you specify multiple SEARCH options, the process must match all the criteria. *criterion* can be any of these:

CAID [*user*]

specifies that the creator accessor ID must match the specified user. If you omit the user identification, #PROCESSINFO searches for a process with the same CAID as the TACL process.

GMOMJOBID [*job-id*]

states that the *job-ancestor.job-id* must match the specified *job-id*. If you omit the job identification, #PROCESSINFO searches for a process with the same GMOMJOBID as the TACL process.

HOMETERM [$*terminal-name*]

specifies that the home terminal name must match the specified $*terminal-name*. If you omit the terminal identification, #PROCESSINFO searches for a process with the same CAID as the TACL process.

HOMETERM [$*terminal-name*]

specifies that the home terminal name must match the specified $*terminal-name*. If you omit the terminal identification, #PROCESSINFO searches for a process with the same HOMETERM as the TACL process.

MINPRI *num*

searches for a process whose priority is greater than or equal to the specified number. (MINPRI can be used in combination with SEARCH PRI to search for a process whose priority lies in a specified range.)

PAID [*user*]

specifies that the process accessor ID must match the specified user. If you omit the user identification, #PROCESSINFO searches for a process with the same PAID as the TACL process.

PRI [num]

specifies that the priority must be less than or equal to the specified num. If you omit the priority specification, #PROCESSINFO searches for a process with a priority less than that of the TACL process.

PROCESSID [[*node-name*.]{$*process-name* | *cpu,pin* }]

specifies that the process identification must match the specified $*process-name* or *cpu,pin* (if you omit both, the process identification must match the TACL process ID).

PROGRAMFILE [[*node-name*.] *file-name-template*]

specifies that the name of the program file must match the specified *file-name-template*. You can include the template characters:

* matches zero or more occurrences of a character

? matches a single occurrence of a character

If you omit the *file-name template*, #PROCESSINFO searches for a process with the same program name as the TACL process.

SUBDEVICE *subdevice-type*

searches for a process whose subdevice type matches the specified *subdevice-type*.

SYSTEM *node-name*

specifies that the named system is to be used instead of the local system.

node-name

is the name of the system on which the process resides.

$*process-name*

is the name of the process or process pair.

cpu,pin

is the CPU number and process identification number for the process.

Result

#PROCESSINFO returns a space-separated list of the information requested.

Considerations

- The information is returned in the order in which you specify the options.

- The process ID that follows the slashes determines where TACL searches for a process:
  - TACL uses a starting CPU,PIN if specified; otherwise, TACL uses the CPU and PIN of its own primary process. TACL starts searching at that CPU and PIN, then searches higher PINs in the same CPU. If no matching process is found, TACL searches higher-numbered CPUs for a matching process.
  - If you specify a $*process-name*, TACL uses the CPU,PIN of the primary process of that process pair as the starting point.
  - If you specify a process pair, TACL returns information about whichever process of the pair, primary or backup, it finds first. If, for example, the primary process of a process pair is running in CPU 4 and the backup process is running in CPU 2, TACL returns information about the backup process if you specify the starting point of the search as CPU,PIN 0,0; it would return information about the primary process if the search were to start at CPU,PIN 3,0.
  - If you specify nothing, TACL starts searching from the CPU,PIN of its own primary process.
- The starting CPU and PIN affect whether #PROCESSINFO returns information about a primary or backup process. TACL returns information about the first matching process that occurs at or beyond the starting point of the search.
- If no process can be found that matches the specifications (RESULT returns a value other than 0 or 1) no other results are returned. Therefore, you should use the RESULT option first.
- If you include one or more SEARCH options, TACL uses the CPU,PIN as the beginning search point.
- If you use SEARCH more than once, the process must meet all the specified criteria.
- To obtain information about a process on a node other than the current default system, specify the remote node by using a SEARCH SYSTEM or SEARCH PROCESSID command (see Examples on page 9-306).

  You can specify the search system redundantly, using both SEARCH and a system specification following the option set, but if you give conflicting system specifications, TACL returns RESULT = 99 (inconsistent parameters).
- Because the home terminal and process-ancestor job ID of a process can reside on different systems from the process itself, specifying a system by either of:

  ```
  #PROCESSINFO /SEARCH HOMETERM \system.$terminal-name/
  #PROCESSINFO /SEARCH GMOMJOBID \system.$process.job-id/
  ```

  has no effect on the search system. Note that it is essential to specify a system in SEARCH HOMETERM or SEARCH GMOMJOBID if the home terminal or process-ancestor job ID is on a system other than the current default system.

- If you do not specify a system in the SEARCH PROGRAMFILE option, TACL does not assume that you are referring to the current default system; instead, it assumes that the program file resides on the search system.
- The recommended way to get information about a given process is as follows:

  ```
  #PROCESSINFO /RESULT, ... ,SEARCH PROCESSID [p]/ 0,0
  ```

  (assuming the variable P contains the process ID). This search method says "search for process [p], starting at CPU,PIN 0,0." Another way of doing this:

  ```
  #PROCESSINFO /RESULT, ... / [p]
  ```

  This second method says "start searching for a process at the CPU,PIN of the primary of [p]," meaning TACL finds [p] itself; but if [p] expands to a CPU,PIN instead of a process name, and if the process at that location has terminated, #PROCESSINFO reports on the process at the next higher CPU,PIN.
- Use care in getting several items of information from one call to #PROCESSINFO, especially if assigning the results to variables with #SETMANY, as some options may return nothing:
  - EXTSWAP is empty if the process does not have an extended swap file.
  - GMOMJOBID is empty if the process has no GMOM.
  - LIBRARY is empty if the process has no library file.
  - MOM is empty if the process is an orphan.
  - SYSTEM is empty if the process is local.

  List those options last to avoid loss of synchronization between destination variables and information items.
- To determine whether a given process exists prior to calling #PROCESSINFO, use the #PROCESSEXISTS built-in function.
- It is possible for a process to have more than one extended data segment. Therefore, repeated calls to #PROCESSINFO with the EXTSWAP option may produce different results, depending on which extended segment the process is using when you request the information. When checking a TACL process, TACLSEGF can be returned as an extended data segment.
- The home terminal and GMOMJOBID for a process can reside on different systems from the target process. The SEARCH HOMETERM and SEARCH GMOMJOBID commands have no effect on the search system.
- If you do not specify a node for SEARCH PROGRAMFILE file or SEARCH PROGRAMFILE template, TACL treats the program file as if it resides on the search system. You can qualify the file or template parameter with a node name, if necessary.
- An OSS program pathname cannot be used as an input file name for a TACL command.

- When retrieving multiple items of information from one call to #PROCESSINFO, some options may return nothing, depending on the software RVU. Use the #SETMANY built-in function carefully. List the potentially empty options last to avoid loss of synchronization between real values and the relative position of these values in the query structure in which they are returned.

Examples

These examples show three different ways to list the CPU number of process $SPLS on system \TEST. $SPLS has CPU and PIN numbers 0,33.

```
15> #PROCESSINFO /SEARCH SYSTEM \TEST, CPU/ $SPLS
#PROCESSINFO expanded to:
0

16> #PROCESSINFO /CPU/ \TEST.$SPLS
#PROCESSINFO expanded to:
0

17> #PROCESSINFO /CPU/ \TEST.0,33
#PROCESSINFO expanded to:
0
```

#PROCESSLAUNCH Built-In Function

Use #PROCESSLAUNCH to start a process. If compared to #NEWPROCESS, #PROCESSLAUNCH enables the use of three additional configuration options for the process being started:

- MAXMAINSTACKSIZE, allowing the user to specify the maximum main stack size
- MAXNATIVEHEAPSIZE, allowing the user to specify the maximum size of the native heap area
- GUARANTEEDSWAPSPACE, allowing the user to specify the amount of space that the process reserves with the Kernel-Managed Swap Facility for swapping.

```
#PROCESSLAUNCH program-file [ / option [ , option ] ... / ]
[ param-set ]
```

program-file

is the name of the file containing the object program to be run. Partial file names are evaluated using the current default node, volume, and subvolume names.

option

is one of these options described for the RUN[D] command:

```
CPU cpu-number
DEBUG
DEFMODE OFF | ON }
EXTSWAP file-name]
GUARANTEEDSWAPSPACE num
HIGHPIN ON | OFF
IN file-name
INLINE
INSPECT OFF | ON | SAVEABEND
INV variable-level [ DYNAMIC [ PROMPT variable-level ] ]
JOBID num
LIB file-name
MAXMAINSTACKSIZE num
MAXNATIVEHEAPSIZE num
MEM num-pages
NAME $process-name
NOWAIT
OUT list-file
OUTV variable-level
PFS num-pages
PRI priority
STATUS variable-level
SWAP file-name
TERM $terminal-name
WINDOW "text"
```

param-set

is a program parameter or a series of parameters sent to the new process in the startup message. Leading and trailing spaces are deleted.

Result

See the #NEWPROCESS Built-In Function on page 9-265.

Considerations

- See the #NEWPROCESS Built-In Function on page 9-265.
- The TACL configuration parameter, CONFIGRUN, must be set to “PROCESSLAUNCH” to enable execution of the #PROCESSLAUNCH code. The three additional options: MAXMAINSTACKSIZE, MAXNATIVEHEAPSIZE, and GUARANTEEDSWAPSPACE, can be specified for the process being created. If CONFIGRUN is not set or is set to “PROCESSCREATE,” these options are ignored.
- For pre-D40 software RVUs, the parameters CURRENTMAINSTACKSIZE, MAXMAINSTACKSIZE, CURRENTNATIVEHEAPSIZE, MAXNATIVEHEAPSIZE, and GUARANTEEDSWAPSPACE have no meaning. If specified, an error message can be returned, indicating the option is invalid.

#PROCESSORSTATUS Built-In Function

Use #PROCESSORSTATUS to determine the status of all CPUs on a given system.

```
#PROCESSORSTATUS [ \node-name ]
```

\node-name

is the name of the system for which you want the processor status. If you omit it, the local system is assumed.

Result

#PROCESSORSTATUS returns a space-separated list of 17 numbers. The first number indicates the highest processor number present in the system, plus one.

The remaining 16 numbers are the status of each of the 16 possible CPUs, starting with CPU 0. For each running CPU, the status is -1. For each halted or absent CPU, the status is 0.

If the specified system is unknown or unavailable, TACL returns this message:

```
Expecting an available system Or End
```

Example

This example illustrates the result of a #PROCESSORSTATUS call:

```
15> #OUTPUT [#PROCESSORSTATUS]
16 -1 -1 -1 -1 -1 -1 0 0 0 0 0 0 0 0 0 0
```

The local system has 16 configured processors and six active processors, running as CPUs 0 through 5.

#PROCESSORTYPE Built-In Function

Use #PROCESSORTYPE to find out the processor type of a specified CPU or of the processor in which a given process is running.

```
PROCESSORTYPE [ / BOTH | NAME / ]
  { [\node-name.]$process-name }  |
  { cpu,pin }  |
  { cpu-num }
```

BOTH

specifies that #PROCESSORTYPE returns both a number and the name of the processor, in text.

NAME

specifies that #PROCESSORTYPE returns the name of the processor, in text.

node-name

is the name of the system on which the specified processor resides.

$*process-name*

is the name of a process running on the CPU whose type is to be reported. The primary CPU of the process is assumed.

cpu,pin

is the CPU number and process identification number for the process.

cpu-num

is the number, 0 through 15, of the CPU whose type is to be reported.

Results

- The #PROCESSORTYPE built-in function returns a name, number, or name and number that indicate the type of the specified processor. If you specify NAME or BOTH, the name of the processor is returned as an 8-character name (as specified in the *Guardian Procedure Calls Reference Manual*). If you specify the BOTH option, TACL returns the number associated with the type of processor, followed by a space and the name of the processor. For a detailed list of processor types and the associated number, see the *Guardian Procedure Calls Reference* manual.

- If an error occurs, the #PROCESSORTYPE built-in function returns a numeric error code as follows:

Code	Meaning
-1	One or more of these: You specified BOTH and the CPU does not exist. You did not specify an option and the CPU does not exist. The system cannot be reached. The process does not exist.
-2	The system does not support the #PROCESSORTYPE built-in function.
-3	You specified the BOTH option, and the system does not support the option. If you specify the NAME option and an error occurs, TACL does not return any text.

Example

If you call #PROCESSORTYPE for CPU 0, and CPU 0 is a NonStop EXT25 processor, TACL returns the following:

```
29> #PROCESSORTYPE 0
#PROCESSORTYPE 0 expanded to:
2
30>
```

#PROMPT Built-In Variable

Use #PROMPT to set or obtain the current state of the prompt flag.

```
#PROMPT
```

Result

#PROMPT returns -1 if the prompt flag is on, 0 if it is off.

Considerations

- When you first log on, #PROMPT is initialized to zero.
- Use #PUSH #PROMPT (or PUSH #PROMPT) to save a copy of the current setting of the prompt flag.
- Use #POP #PROMPT (or POP #PROMPT) to restore the prompt flag from the last copy pushed.
- Use #SET #PROMPT (or SET VARIABLE #PROMPT) to set the prompt flag off or on.

  The syntax of #SET #PROMPT is:

  ```
  #SET #PROMPT num
  ```

  num

  is zero to set the flag off, a nonzero value to set it on.

  - When it is on, the prompt flag causes TACL to try to invoke a macro or routine called _PROMPTER just before issuing a prompt. Your _PROMPTER macro can add text to the standard TACL prompt by using the #PREFIX built-in variable.
  - If you press BREAK or an error occurs during your prompt macro, #PROMPT is automatically set to zero.

#PURGE Built-In Function

Use #PURGE to remove a file from a disk.

```
#PURGE file-name
```

file-name

is the name of the file to be removed.

Result

#PURGE returns zero if it removes the file successfully; otherwise, it returns the file-system error indicating the reason for the failure.

Considerations

- When you use #PURGE to remove a disk file, the file entry is removed from the file directory in that volume, and any space previously allocated to that file is made available. Data in the file is not physically removed from the disk unless you specified the CLEARONPURGE option when you created the file: removed files are then overwritten with spaces. For information about the
- CLEARONPURGE option, see the FUP CREATE command description in the *File Utility Program (FUP) Reference Manual*. You can purge a file only if it is not currently open. You must have purge access to the file. See the description of the FUP SECURE command in the *File Utility Program (FUP) Reference Manual* for information about file-access restrictions.
- If you try to use the PURGE command to remove a file that is being audited by the TMF subsystem, the attempt fails, and file-system error 12 (file in use) is returned if there are pending transaction-mode records or file locks. A PURGE attempt of this kind is blocked regardless of whether the processes that opened the file still exist.

Example

This example illustrates the use of #PURGE with status reporting:

```
#PUSH old^file
#SET old^file $DATA.FILES.TEMP1
[#IF [#PURGE [old^file]] |THEN| #OUTPUT **Purge failed**]
```

#PUSH Built-In Function

Use #PUSH to create a new top-level definition for one or more variables or built-in variables.

```
#PUSH variable [ [,] variable ] ...
```

variable

is a valid variable name or a built-in variable name.

Result

#PUSH returns nothing.

Considerations

- The PUSH command is an alias for the #PUSH built-in function and can be used interchangeably with it.
- For variables, #PUSH creates an empty top level of type TEXT.
- For built-in variables, #PUSH creates a new top-level definition, copying the old top-level definition to the new one (top-level and second-level definitions are now the same).
- If the variable does not exist, PUSH registers the name of the variable, but does not allocate space until you use #SET or a similar command or built-in function to actually place data into the variable. As a result, you must perform a SET VARIABLE or related operation prior to using the variable in an #IF call or other command or function that tests the value of the variable.
- Do not try to #PUSH the root directory (:). If you try this, TACL returns "*ERROR* Cannot push or pop the root segment's root."
- To avoid losing standard functionality from your TACL environment, do not #PUSH directories supplied as part of the TACL product (such as UTILS).

#RAISE Built-In Function

Use #RAISE in a routine to explicitly cause an exception. The exception must be filtered by some active routine.

```
#RAISE exception
```

exception

can be any of these:

_BREAK

indicates that the BREAK key has been pressed.

_ERROR

indicates that an error has occurred.

user_exception

is any text, 1 to 31 alphanumeric characters in length, of which the first must be alphabetic.

Result

#RAISE returns nothing.

Consideration

All routines up to and including the one having the exception filter are eliminated, and the routine that issued #FILTER is reinvoked. You can use #EXCEPTION to deter mine why the routine was reinvoked. (See the discussion of exception handling in the *TACL Programming Guide* and the example fro the #FILTER Built-In Function on page 9-178.)

#RENAME Built-In Function

Use #RENAME to change the name of an existing disk file.

```
#RENAME old-file-name new-file-name
```

old-file-name

is the name of the disk file to be renamed.

new-file-name

is the new name for the file.

Result

#RENAME returns zero if it renames the file successfully otherwise, it returns the file-system error that indicates the reason for the failure. If, however, the error is caused by a missing *old-file-name*, or if that file does not exist, TACL displays an "`expecting the name of an existing file`" message.

Considerations

- The new file name must be on the same volume as the old file name.
- You can rename a file only if it is not open with exclusive access and you either have purge access to the file or are logged on as a super-group user.
- You can use the RENAME command to change the subvolume name of a file, but not its volume name. Disk files that are renamed stay on the same disk volume.
- To change the volume where a file resides, copy the file to a new volume with the FUP DUP command, then delete the original file. For details, see the *File Utility Program (FUP) Reference Manual*.
- If you try to rename a file being audited by the TMF subsystem, the attempt fails and file-system error 80 (operation invalid) is returned.
- Both format 1 and format 2 files can be renamed.

Example

This code renames a file and checks whether the rename was performed without errors:

```
[#IF [#RENAME oldn newn] |THEN|
#OUTPUT Rename Error
]
```

#REPLY Built-In Function

Use #REPLY to add text to the reply if the TACL process IN file is $RECEIVE.

```
#REPLY [ text ]
```

text

is the text to be added to the reply.

Result

#REPLY returns nothing.

Considerations

- The #REPLY function is used when TACL functions as a server. For additional information, see the #SERVER Built-In Function on page 9-339 built-in function.
- If there is already text in the reply, through previous calls to #REPLY or #REPLYV (or calls to #OUTPUT or #OUTPUTV if the OUT file for TACL is also $RECEIVE), a space precedes the added text in the accumulated reply.

#REPLYPREFIX Built-In Variable

Use #REPLYPREFIX to examine the current value of the reply prefix, which can be an integer or empty. The #REPLYPREFIX function adds a prefix to a reply when TACL functions as a server. For additional information about TACL as a server and an example showing the use of #REPLYPREFIX, see the *TACL Programming Guide*.

```
#REPLYPREFIX
```

Result

#REPLYPREFIX returns the current value of the reply prefix.

Considerations

- The #REPLYPREFIX function is used when TACL functions as a server. For examples of TACL servers, refer to the *TACL Programming Guide*.
- When you first log on, #REPLYPREFIX is initialized to a null value.
- TACL never changes the value of #REPLYPREFIX unless instructed to do so.
- When a Pathway requester sends a message to a server, the server must return a reply message that includes a reply prefix. The server can use this reply prefix to indicate what is in the rest of the message. When the requester receives the message, it can extract information as indicated by the value of the reply prefix.
- Use #PUSH #REPLYPREFIX (or PUSH #REPLYPREFIX) to save a copy of the current reply prefix.
- Use #POP #REPLYPREFIX (or POP #REPLYPREFIX) to restore the reply prefix from the last copy pushed.
- Use #SET #REPLYPREFIX (or SET VARIABLE #REPLYPREFIX) to set an unsigned 16-bit integer reply prefix. If TACL is in server operation and #REPLYPREFIX contains a 16-bit integer, each reply to an unqualified (control) opener of TACL is prefixed by that number.

  The syntax of #SET #REPLYPREFIX is:

  ```
  #SET #REPLYPREFIX [ num ]
  ```

  num

  is a 16-bit integer that can be decoded by a Pathway SEND statement. If you omit num, you do not have a reply prefix.

#REPLYV Built-In Function

Use #REPLYV to add a copy of the text in a string to the reply if the TACL IN file is $RECEIVE. You can also use #REPLYV to append the binary data of a STRUCT to the reply when TACL is operating as a server.

```
#REPLYV string
```

string

is the name of an existing variable level, which can be of type STRUCT, or text enclosed in quotation marks, or a concatenation of such entities. The concatenation operator is '+' (the apostrophes are required).

Result

#REPLYV returns nothing.

Considerations

- The #REPLYV function is used when TACL functions as a server. For additional information, see #SERVER Built-In Function on page 9-339.
- If there is already text in the reply, through previous calls to #REPLY or #REPLYV (or calls to #OUTPUT or #OUTPUTV if the TACL OUT file is also $RECEIVE), a space precedes the added text in the accumulated reply.
- The form

  ```
  #REPLYV struct
  ```

is not equivalent to

```
#REPLY [ struct]
```

The latter appends the external representation of the STRUCT, rather than its actual binary data, to the reply.

Example

This example shows the use of both quoted text and a variable name in constructing a reply:

```
#PUSH termname
#SET termname [#MYTERM]
#REPLYV "My terminal is " '+' termname '+' " at this time."
```

Assuming that the home terminal is named $WEIRD at the time the #SET function is invoked, the following is included in the reply:

```
My terminal is $WEIRD at this time.
```

#REQUESTER Built-In Function

Use #REQUESTER to open and close processes, devices, and structured and unstructured files and to read and write those files.

```
#REQUESTER [ / option [ , option ] / ]
  {
    { CLOSE variable-level } |
    { READ file-name error-var read-var prompt-var } |
    { WRITE file-name error-var write-var }
  }
```

option

specifies one of these:

`EXCLUSION { SHARED | PROTECTED | EXCLUSIVE }`

specifies the type of file sharing to be used with the READ or WRITE creation parameter.

`EXCLUSIVE`

specifies that others cannot open *file-name*.

`PROTECTED`

specifies that others can open *file-name* for reading, but not for writing.

`SHARED`

specifies that others can open *file-name* for reading or writing. This option is ignored if used with the CLOSE parameter. If omitted, the default parameter for READ is SHARED, and for WRITE is PROTECTED for disk files.

`WAIT [ num ]`

specifies that the requester is to do waited I/O rather than no-wait I/O (the default). With waited I/O, TACL waits automatically as needed: TACL executes subsequent functions after initiating an I/O request, but if it encounters an #APPEND, #APPENDV, #EXTRACT, or #EXTRACTV function that refers to a requester variable, it stops until the pending I/O operation is finished.

If you do not specify the WAIT option, TACL continues execution immediately after placing an I/O request. After allowing execution of independent functions, use the #WAIT function on the requester variable to ensure that the I/O operation is complete before you access the variable again.

A requester normally uses a buffer length of 239 characters, but you can exceed that limit by specifying num as a value in the range 132 to 5000. If you omit num, the requester does waited I/O, but the 239-character default buffer size remains in effect.

CLOSE *variable-level*

closes the file and deletes the requester associated with *variable-level*. If it is a write requester, use #WAIT on *write-var* to ensure that all data in the write variable has been written to the file before you delete the requester.

variable-level

is any one of the variable levels used when #REQUESTER was invoked to open a file. For example, if you start a requester with the READ option, you can specify the same *error-var*, *read-var*, or *prompt-var* with the CLOSE option. A given variable level can be associated with only one requester or server at any time. The variable level must not be a DIRECTORY, STRUCT, or a STRUCT item.

READ *file-name error-var read-var prompt-var*

creates a read requester and opens *file-name* for reading. For each line appended to *prompt-var*, the requester reads a record from *file-name* and appends it to *read-var*. If *file-name* is a terminal or a process, appending a line to *prompt-var* causes the value of the variable to be transmitted as a prompt. If you use the WAIT option, TACL begins executing subsequent code, but if it encounters #APPEND(V) or #EXTRACT(V) that refers to a requester variable, it waits until the I/O operation is finished.

If you do not use WAIT, TACL continues execution. To avoid reading incorrect data, use the #WAIT function. Use #APPEND(V) *prompt-var* and #EXTRACT(V) *read-var* instead of the #SET(V) built-in functions. If an error occurs, *error-var* is set to the error number and further I/O operations on the requester stop until you clear *error-var* by setting it to a null value.

file-name

is the name of the file to be read. If the file does not exist, an error occurs. Both format 1 and format 2 files (unstructured, file code 0 format 2 files only) can be read.

error-var

is the name of a variable level to hold error codes.

read-var

is the name of a variable level to hold the data read.

prompt-var

is the name of a variable level to hold one prompt for each line to be read.

Note. The variable level cannot be a DIRECTORY, STRUCT, or STRUCT item.

WRITE *file-name error-var write-var*

creates a write requester and opens *file-name* for output. For each line added to *write-var* with #APPEND(V), the requester writes a record containing that line to *file-name*.

If you include the WAIT option, TACL begins executing subsequent code, but if it encounters #APPEND(V) or #EXTRACT(V) that refers to a requester variable, it waits until the I/O operation is finished.

If you do not use WAIT, TACL resumes execution. To avoid writing over valid data, use #WAIT *write-var*; use #APPEND(V) *write-var* instead of using #SET(V).

If an error occurs, *error-var* is set to the error number and further I/O operations on the requester stop until you clear *error-var* by setting it to a null value.

file-name

is the name of the file to be written to. If *file-name* does not exist, TACL creates an edit-format file; if it does exist, data is appended to the end of the file.

error-var

is the name of a variable level to hold error codes.

write-var

is the name of a variable level containing the data that is to be written.

△ **Caution.** TACL truncates data appended to a write variable associated with a #REQUESTER, but does not return an error.

Result

#REQUESTER returns zero if it is successful; otherwise, it returns a file-system error indicating the reason for failure. A TACL error causes an exception rather than returning an error.

Considerations

- The #REQUESTER function does not create a separate requester function. It runs as part of the TACL process. You read and write data through the use of associated variables (*read-var*, *prompt-var*, and *write-var*).
- The first call to #REQUESTER opens the file and associates the variables with the file, but does not perform any input or output to the file.
- To initiate input for a READ operation, place data in *prompt-var* (in this example, defined as pvar):

```
#APPEND pvar *read next*
```

This action triggers a read operation. If you are reading from a file, TACL ignores the contents of *prompt-var*. If you are reading from a process or device, TACL sends the contents of *prompt-var* to the process or device as part of a WRITEREAD operation.

- If you specify a WRITE requester for a nonexistent disk file, TACL creates an edit-format file; if you specify a READ requester for a nonexistent file, a file-system error occurs.
- If #REQUESTER is unsuccessful in creating a requester, such as in the case above, it returns the file-system error in its result. If it is successful, any future errors (such as end-of-file in a read requester) are returned in *error-var*.
- All I/O for no-wait requesters takes place while normal TACL processing continues. I/O automatically occurs when information becomes available in the variables and files specified. To synchronize I/O, use the #WAIT built-in function.
- A requester, waited or no-wait, that does I/O on a disk file requires that TACL internally allocate one of its block buffers. Because these block buffers are large and must be allocated from the first 64K bytes of the TACL address space, there are only four of them. Each block buffer is 1024 bytes. The #IN and #OUT built-in functions also require these buffers. A block buffer is passed to the SIO procedure to help perform read and write operations to the file. SIO uses the block buffers whenever necessary.
- For structured files, block buffers can be used to improve the efficiency of read operations. For edit-format files, block buffers are required by SIO to perform write padding of records. For other types of files, block buffers can be used for record blocking and unblocking.
- If you specify a waited requester with a record length greater than 1024 bytes, TACL allocates one of its block buffers. SIO automatically turns off block buffering because the record length is greater than the block buffer length.
- The only way you can use large buffers is by using waited requesters; a waited requester still counts against the block buffer limit.
- The maximum record lengths are as follows:

Edit-format files	239 bytes
No-waited requesters	239 bytes
Unstructured non-edit files	1024 bytes
Other waited requesters	5000 bytes

- For non-edit disk files, trailing blanks at the end of a STRUCT (for example, FILLER bytes or trailing blanks in the last STRUCT data item) are not trimmed from the output record before being written to the file using the #APPENDV built- in function. For edit-format files, SIO automatically trims trailing blanks from the output record before it is written to the file.

- If the data to be written to an unstructured non-edit file is smaller than the maximum record length, the remaining bytes of the record are padded with trailing blanks to fill out the logical record.
- When communicating with a process, be careful to coordinate functions that enable the communication (such as #REQUESTER) with the counterpart mechanisms in that process (such as IN or OUT referring to the TACL process name). Deadlock conditions can result if TACL tries to open a process for communication at the same time that process is trying to open TACL for communication.
- For additional information about using #REQUESTER to interact with a TACL process that uses #SERVER, see #SERVER Built-In Function on page 9-339.

#RESET Built-In Function

Use #RESET to reset the argument pointer, frame number, reply value, and result text.

```
#RESET option [ option ] ...
```

option

is a reset request. It can be any of these:

ARGUMENTS

sets the argument pointer (see #ARGUMENT Built-In Function on page 9-21) to the beginning of the routine argument list. This option is applicable only within a routine.

FRAMES

in a routine, invokes #UNFRAME for all frames with frame numbers higher than they were when the routine was entered. Outside of a routine, it invokes #UNFRAME for all frames with numbers higher than they were when the last prompt was issued.

REPLY

discards the reply thus far accumulated. For additional information, see #REPLY Built-In Function on page 9-317, #REPLYPREFIX Built-In Variable on page 9-318, and the description in the *TACL Programming Guide* of TACL operating as a server.

RESULTS

discards all previous results of #RESULT. This option is applicable only from within a routine.

Result

#RESET returns nothing.

#REST Built-In Function

Use #REST to examine the unprocessed portion of the argument set of the current routine. #REST does not process the remainder of the argument.

```
#REST
```

Result

#REST returns the remainder of the arguments of the current routine.

Considerations

- An internal end-of-line is a binary zero and is not converted to a space.
- #REST does not advance the current-position pointer.
- To parse a set of arguments passed to a routine, use the #ARGUMENT built-in function.
- To specify the position at which the next #ARGUMENT function is to resume processing arguments, use the #SETSCAN built-in function.
- To obtain the number of characters that #ARGUMENT has processed, use the #GETSCAN built-in function.

#RESULT Built-In Function

Use #RESULT to supply the text that is to replace the original invocation of a routine.

```
#RESULT [ text ]
```

text

specifies the text that replaces the original invocation of the routine in which this function appears.

Result

#RESULT returns nothing.

Considerations

- If your routine contains no calls to #RESULT, the routine returns nothing.
- If your routine contains multiple calls to #RESULT, the text supplied by each call is separated by spaces from the text supplied by the preceding call to #RESULT.
- A routine can invoke #RESULT at any time during its execution.
- A routine can discard all previous results by invoking #RESET RESULTS.

#RETURN Built-In Function

Use #RETURN to exit immediately from a routine.

```
#RETURN
```

Result

#RETURN returns nothing.

Considerations

- TACL immediately exits from the current routine as though there were no more code after #RETURN.
- #RETURN does not reset #FRAMEs. If you use #RETURN, be sure to #UNFRAME any #FRAME within your routine or call #RESET FRAMES to invoke #UNFRAME for all frames with frame numbers higher than they were when the routine was entered.
- To provide alternative exits, you can include multiple, conditional #RETURN commands in a single routine.

#ROUTEPMSG Built-In Variable

The built-in variable #ROUTEPMSG provides the capability to suppress system and process messages. Use #ROUTEPMSG to set or obtain the current state of the #ROUTEPMSG built-in variable.

```
#ROUTEPMSG { ALL | STANDARD |
( message-type [ message-type ] ... ) }
```

ALL

suppresses all messages selected by the #PMSG built-in variable setting from being output.

STANDARD

no suppression occurs. Messages selected by the #PMSG built-in variable setting are output to the current OUT (default value).

message-type

can be any of these:

SYSTEM

suppresses system messages from being output. Note that the #PMSG built-in variable setting does not matter, because system messages are output regardless of the #PMSG built-in variable setting.

NORMAL

suppresses normal process messages, if selected by the #PMSG built-in variable setting, from being output.

ABNORMAL

suppresses abnormal process messages, if selected by the #PMSG built-in variable setting, from being output. Like all other TACL built-in variables, if a backup CPU is specified for the TACL process and the primary TACL process fails, the backup TACL does not inherit the TACL variables from the primary TACL.

Considerations

- Use caution when suppressing messages. If messages are suppressed, include TACL statements that detect abnormal completion of processes by checking the status of the TACL built-in functions and the process completion variables (:_completion or :_completion^procdeath). On an abnormal run of TACL programs, rerun the TACL programs with messages on.

- The output of certain message classes is determined by the #PMSG built-in variable setting. The #ROUTEPMSG built-in variable allows specified message classes to be suppressed from being output.
- When you first log on, #ROUTEPMSG is initialized to STANDARD.
- Use #PUSH #ROUTEPMSG (or PUSH #ROUTEPMSG) to save a copy of the current setting of the #ROUTEPMSG built-in variable.
- Use #POP #ROUTEPMSG (or POP #ROUTEPMSG) to restore the #ROUTEPMSG built-in variable from the copy last pushed.
- Use #SET #ROUTEPMSG (or SET VARIABLE #ROUTEPMSG) to set the #ROUTEPMSG built-in variable.
- The classes of message affected by the #ROUTEPMSG built-in function are:

System message	CPU down, CPU up, and so forth
Normal process message	Process start, stop, and so forth
Abnormal process message	Abort process, backup died, and so forth

- These tables define the outcome of setting the #PMSG and #ROUTEPMSG built-in variables. These outcomes are possible:

Yes	Message is output to the current OUT file.
No	Message is not output.
N.A.	Message is not output due to the #PMSG setting.

- This table defines the type of messages output by the #PMSG setting:

#PMSG	System	Normal Process	Abnormal Process
-1 (on)	Yes	Yes	Yes
0 (off)	Yes	No	Yes

- This table defines how the #ROUTEPMSG setting affects message output if #PMSG is on:

#ROUTEPMSG	System	Normal Process	Abnormal Process
ALL	No	No	No
SYSTEM	No	Yes	Yes
NORMAL	Yes	No	Yes
ABNORMAL	Yes	Yes	No
SYSTEM NORMAL	No	No	Yes

#ROUTEPMSG	System	Normal Process	Abnormal Process
SYSTEM ABNORMAL	No	Yes	No
NORMAL ABNORMAL	Yes	No	No
STANDARD	Yes	Yes	Yes

- This table defines how the #ROUTEPMSG setting affects message output if #PMSG is off:

#ROUTEPMSG	System	Normal Process	Abnormal Process
ALL	No	N.A.	No
SYSTEM	No	N.A.	Yes
NORMAL	Yes	N.A.	Yes
ABNORMAL	Yes	N.A.	No
SYSTEM NORMAL	No	N.A.	Yes
SYSTEM ABNORMAL	No	N.A.	No
NORMAL ABNORMAL	Yes	N.A.	No
STANDARD	Yes	N.A.	Yes

Example

To suppress all messages, enter:

```
> #SET #ROUTEPMSG ALL
> #ROUTEPMSG
```

#ROUTEPMSG returns the current state of the #ROUTEPMSG built-in variable, which could be one of:

```
ALL
SYSTEM
NORMAL
ABNORMAL
SYSTEM NORMAL
SYSTEM ABNORMAL
NORMAL ABNORMAL
STANDARD
```

#ROUTINENAME Built-In Function

Use #ROUTINENAME to allow a routine to obtain its own name; the routine can then use this information to invoke itself.

```
#ROUTINENAME
```

Result

#ROUTINENAME returns the fully qualified name of the variable level holding the currently active routine. If invoked from a routine stored in a ?TACL ROUTINE file, it returns the name of the variable TACL created to hold a copy of the routine while it is active (it cannot obtain the name of the original file).

Considerations

- #ROUTINENAME performs a similar function in TACL routines as %0% performs in TACL macros (see the detailed descriptions in the *TACL Programming Guide* of the differences and similarities between macros and routines). The two features are not interchangeable, however, as shown by this example.
- To invoke the entire routine from within the routine, obtain the routine name by calling the #ROUTINENAME built-in function.

Examples

Given this library,

```
?SECTION rumph ROUTINE
#OUTPUT Percents show name as %0%.
#OUTPUT Routinename shows name as [#routinename].

?SECTION grumph MACRO
#OUTPUT Percents show name as %0%.
#OUTPUT Routinename shows name as [#routinename].
```

invoking the routine and the macro gives these results:

```
36> RUMPH
Percents show name as %0%.
Routinename shows name as :RUMPH.1.
37> GRUMPH
Percents show name as :GRUMPH.1.
#OUTPUT Routinename shows name as [#routinename]
                                  ^
*ERROR* No routine has been called
```

You can also use #ROUTINENAME to invoke a variable that will reside in the same directory as the routine itself, without stating the directory name explicitly:

```
[#VARIABLEINFO /DIRECTORY/ [#ROUTINENAME]]:elf elf-args
```

#SEGMENT Built-In Function

Use #SEGMENT to determine the name of the file TACL is using as its default segment file to hold its variables.

```
#SEGMENT [ / USED / ]
```

USED

requests an estimate of the number of bytes currently used in your segment.

Result

Without the USED option, #SEGMENT returns the segment file name.

If you include the USED option, #SEGMENT returns an estimate of the number of bytes currently used in your segment.

Consideration

To obtain information about a segment file, use the #SEGMENTINFO built-in function.

#SEGMENTCONVERT Built-In Function

#SEGMENTCONVERT converts a segment file from a C00/C10 format version to a C20 or later version, or the reverse. Segment files created under newer RVUs of TACL have a different format from those created under earlier RVUs of TACL because of character changes necessitated by the addition of the international character set with the C20 RVU.

```
#SEGMENTCONVERT / FORMAT { A | B } /
  old-file-name new-file-name
```

FORMAT

specifies the format in which *new-file-name* is to be created.

FORMAT A specifies that *new-file-name* is a C00/C10 format version.

FORMAT B specifies that *new-file-name* is a format version C20 or later.

old-file-name

is the name of an existing TACL segment file whose format is to be converted.

new-file-name

is the name to be given to the new segment file that #SEGMENTCONVERT creates. #SEGMENTCONVERT creates a new segment file, *new-file-name*, in the format specified by the FORMAT argument, and copies the contents of the existing segment file, *old-file-name*, to it.

Result

#SEGMENTCONVERT returns nothing.

Considerations

- If *old-file-name* does not exist, cannot be opened, cannot be read, or is not a segment file (file code 440), or if *new-file-name* already exists, TACL issues an appropriate error message.
- If the format specified for *new-file-name* is the same as that of *old-file-name*, #SEGMENTCONVERT only duplicates the file.
- It is possible to use CREATESEG and the same LOAD or #LOAD operations that originally built an older segment file to re-create the file in the newer format; but if you no longer have the source code that created the segment file, use #SEGMENTCONVERT to convert the file.
- It is a good idea to convert C00-format or C10-format segment files for use on newer systems.

  - Whenever a RVU of TACL after C10 tries to attach a C00/C10 segment file, it automatically creates a temporary segment file in format version C20 or later, copies the older file to it, and attaches the temporary file instead. This typically happens every time you log on. Converting the older file to the newer format eliminates this action, as well as the disk space requirements of the temporary file.
  - If you create a variable with a C20 or later RVU of TACL, and the variable contains a “y-dieresis” character (in the international character set), and you write the variable to a C00/C10 segment file, it can corrupt the file. Converting the file to newer format eliminates that risk.

- Conversely, a newer segment file cannot be attached by a C00 or C10 RVU of TACL. #SEGMENTCONVERT can convert such a file to the older format. To obtain the version of a segment file, use the #SEGMENTVERSION built-in function.
- You cannot attach more than 50 segment files.

#SEGMENTINFO Built-In Function

Use #SEGMENTINFO to obtain information about any of the segments being used by your TACL process.

```
#SEGMENTINFO / option [ , option ] / [ segment-id ]
```

option

is an information request. It can be any of these:

ACCESS

returns the access mode of the segment, SHARED or PRIVATE.

DIRECTORY

returns the directory in which the segment file appears.

FILENAME

returns the name of the current segment file.

ID

returns the segment ID for which information is being returned.

PRIMARY

returns the primary extent size of the segment file.

RESULT

returns 0 if the segment ID for which information is sought is the one specified in the argument, 1 if the information is for the next higher segment ID in use, or 2 if there is no higher segment ID in use.

SECONDARY

returns secondary extent size of the segment file.

USECOUNT

returns the number of pointers in your TACL process that point into the segment in question. Other TACL processes are not included. Pointers result from using variables in the segment:

- As the home directory
- In the use list
- For requesters
- For servers (both explicit and implicit)

- For execution
- To point to other segments

USED

returns the number of bytes in use in the segment.

segment-id

is the segment ID at which to begin searching. If you omit it, segment zero is assumed.

Result

#SEGMENTINFO returns a space-separated list of the specified information in the order it was requested.

Consideration

To obtain the name of your default segment file, use the #SEGMENT built-in function.

#SEGMENTVERSION Built-In Function

#SEGMENTVERSION returns the format (C00/C10 or C20 or later) of a segment file.

```
#SEGMENTVERSION file-name
```

file-name

is the name of an existing TACL segment file.

Result

#SEGMENTVERSION returns one of these characters:

A if the segment file is in C00/C10 forma

B if the segment file is a format version C20 or later

If the specified file does not exist, cannot be opened, cannot be read, or is not a TACL segment file (file code 440), TACL issues an appropriate error message.

Consideration

To convert a segment to the other format, use the #SEGMENTCONVERT built-in function.

#SERVER Built-In Function

The #SERVER built-in function allows you to use your TACL process to interact with one or more other processes. This mechanism is an alternative to the use of INV and OUTV or the use of INLINE commands. Processes open your TACL process as if it were a file. Your TACL process can then read requests and respond to them. You can control processes, such as utilities, and interpret their responses.

The name defined by #SERVER remains in existence until you delete it with another call to #SERVER (with the KILL option).

Your TACL must have a process name for you to be able to use #SERVER.

```
#SERVER / option [ , option ] ... / [ name ]
```

option

can be any of these:

IN *variable-level*

specifies a variable level to be used to supply data to WRITEREAD procedures called by requesters. The first line of the variable level is removed and given to each WRITEREAD. If the variable level is empty, WRITEREAD waits until data is placed into the variable level. To avoid race conditions (see Considerations on page 9-340), use #APPEND or #APPENDV to add data to this variable level. You cannot read data out of this variable level, as the data might be passed immediately to a process and therefore be no longer available.

To send an end-of-file, use #EOF. If the IN variable is empty and a process is waiting, TACL sends an end-of-file immediately; otherwise, TACL sets a flag. The next time the process tries to read from the empty server file, TACL sends an end-of-file and clears the flag. #EOF does not alter the state of the server file or the IN variable, but it can cause the process to terminate, which can alter the state of the server file.

OUT *variable-level*

specifies the variable level to be used to accept data from WRITE procedures called by requesters. WRITEs are appended one line at a time to the variable level. To avoid race conditions (see Considerations on page 9-340), use #EXTRACT or #EXTRACTV to obtain data from this variable level. You must not read and then clear this variable level as more data can arrive between the read and the clear, and that data is then lost.

PROMPT *variable-level*

specifies the variable level that is to contain the most recent data sent by a WRITEREAD. Previous data in the variable level is lost each time a WRITEREAD occurs.

KILL

specifies that the indicated server is to be deleted. You must specify a server-name for this option. If a server is deleted, any processes that still have the server open receive file-system error 66 (device downed) on subsequent I/O operations to the named server.

name

is the access name of your TACL process. This argument is optional unless you specify the KILL option. The name consists of your TACL process name followed by a period, a number sign, a letter, and zero to six alphanumeric characters ($D127.#K39, for example); if you want, that can be followed by another period, a letter, and zero to seven alphanumeric characters ($D127.#K39.ACM48, for example). If you specify a create option and do not specify a name, TACL creates one, of the form:

$*your-tacl-name*.#S*nn*

where *nn* is the next available server number.

Result

- If you supply one or more create options, #SERVER returns a name for your TACL process.
- If you specify the KILL option, #SERVER returns nothing.

Considerations

- When a process writes to your TACL process, the line is appended to the end of the OUT variable of the server. When a process prompts for input, the prompt is stored in your PROMPT variable, destroying all previous contents. When a process reads a line from your TACL process, TACL removes the first line of the IN variable and passes it to the process. If the IN variable is empty, the requesting process waits until you put more data into the IN variable.
- A TACL process can have a maximum of 100 simultaneous openers, but only one process can read from it at a time. If one process has requested data from the IN variable and a second process tries to retrieve data from the IN variable, the second process receives a file-system error 28 (attempt to open a disk file or $RECEIVE with maximum number of concurrent operations greater than one).
- You can use the create options-IN, OUT, and PROMPT-in combination, but KILL must be used alone.
- There is a potential deadlock situation when using #SERVER to control DEBUG and INSPECT for a process started by the same TACL. The deadlock occurs because TACL waits for the program to read its startup messages, which it cannot do because it is waiting for an R command from DEBUG or INSPECT, which in turn is waiting for input from a TACL variable. To avoid this situation, before issuing

#SERVER, initialize the IN variable for #SERVER to contain enough DEBUG or INSPECT commands to ensure that the program gets past the point where it reads its startup messages.

- When communicating with a process, be careful to avoid race conditions. Coordinate functions that enable communication (such as #SERVER) with counterpart mechanisms in that process. Deadlock conditions can result if TACL tries to open a process for communication at the same time that process is trying to open TACL for communication.
- If TACL is unable to allocate space in an OUT or PROMPT variable, the process doing the WRITE or WRITEREAD receives a file-system error 45 (file is full).
- If a server name is eliminated, any variables associated with the server remain the same. Any processes that still have it open receive a file-system error 66 (device downed) on subsequent I/O operations using the server name.
- To create a server implicitly, run a program and supply and accept data using INV or OUTV. For additional information, see #NEWPROCESS Built-In Function on page 9-265 or the RUN[D|V] Command on page 8-156. An implicitly created server remains in existence until its parent process terminates, or you can delete it using #SERVER with the KILL option.
- If you log off, all server files are deleted immediately.
- To synchronize processes that use your TACL server, use the #WAIT built-in function.

Examples

To create an access name for your TACL process:

```
#PUSH access_name in_var out_var prompt_var
#SET access_name [#SERVER /IN in_var, PROMPT prompt_var, &
OUT out_var/]
```

Table 9-12 lists commands that a TACL process using #REQUESTER and a TACL process using #SERVER can use to communicate with each other. The decision to use one-way or two-way communication depends entirely upon how the server responds to requests.

Table 9-12. Communicating with a TACL Requester

Type of Communication	#Requester	#Server
Two-way communication	Invokes #REQUESTER with the READ operation to open the process	
(WRITEREAD operation)	Invokes #APPENDV *prompt^var var* (initiates a WRITEREAD operation)	Invokes #EXTRACTV *prompt^var var* (performs a READUPDATE operation)
	Invokes #EXTRACTV *read^var data* (retrieves the response)	Invokes #APPENDV *in^var data* (initiates a REPLY operation)
One-way communication	Invokes #REQUESTER with the WRITE option to open the process	
(WRITE operation)	Invokes #APPENDV *write^var var* (initiates a WRITE operation)	Invokes #EXTRACTV *out^var data* (performs a READ operation)

#SET Built-In Function

Use #SET to change the contents of a variable level or built-in variable.

```
#SET
  { [/ option [, option ]/] variable-level [ text ] } |
  { built-in-variable [ built-in-text ] }
```

option

is either of these:

IN *file-name*

specifies that the named file is to be read into the variable level. If this option is present, text is not allowed.

TYPE *type-name*

specifies the type of the variable level being set.

type-name

is one of these:

ALIAS

specifies that *variable-level* is an alias; therefore, text must be a *variable-level* name or a file name.

DELTA

specifies that *variable-level* is a #DELTA command variable; therefore, text must be #DELTA commands.

DIRECTORY

specifies that *variable-level* is a directory name. If you omit text, this option clears *variable-level* and establishes an empty directory.

If you include text, it must have this specific form:

mode *file-name*

where mode is either PRIVATE or SHARED, and *file-name* is the name of an existing segment file (code 440). The segment file must not reside on a remote system. This form of the #SET command associates a directory variable with the segment file in the same manner as an ATTACHSEG command.

MACRO

specifies that text is a TACL macro.

ROUTINE

specifies that text is a TACL routine.

TEXT

specifies that text is simply text; that is, it has no special meaning to TACL.

variable-level

is the name of the existing variable level to be set.

text

is the text to be put into the specified variable level.

built-in-variable

is the name of a built-in variable.

built-in-text

is the new value for the built-in variable.

Result

#SET returns nothing.

Considerations

- #SET replaces the current contents of *variable-level* with the specified text or, if you indicate a file with the IN option, with the contents of the specified file.
- You cannot use a text entry with the IN option, nor can you use the IN option with a built-in variable.
- The IN option reads data in the PLAIN mode. The TACL process does not interpret metacharacters as metacharacters; nor does it expand variables.
- Unless you specify a TYPE option, the type of the variable level remains unchanged. You cannot use the TYPE option with a built-in variable.
- When setting a built-in variable, the format of text must be appropriate for that particular variable. See the description of the built-in variable in question.
- You cannot insert leading or trailing spaces into a variable level with #SET.
- To copy a string to an existing variable level, use the #SETV built-in function.

- To distribute the members of a space-separated list into several individual variable levels, use the #SETMANY built-in function.
- You cannot attach more than 50 segment files.

Examples

1. This example sets a built-in variable:

```
32> #SET #WIDTH 132
```

2. The next example sets a user-defined variable level:

```
33> #SET vara This is it
```

3. This example illustrates the use of level specifications to put different information into different levels of the variable var:

```
34> #PUSH var
35> #PUSH var
36> #SET / TYPE TEXT / var.1 TIME
37> [#SET / TYPE MACRO / var.2
37> #OUTPUT The first argument is %1%
37> ]
38> var.1
July 12, 1990 15:55:51
39> var.2 hardest
The first argument is hardest
```

#SETBYTES Built-In Function

#SETBYTES moves data between STRUCTs. Use #SETBYTES to fill one STRUCT or STRUCT item with as many copies of the data from another STRUCT or STRUCT item as can fit, without regard to the data types of the fields involved.

```
#SETBYTES destination source
```

destination

is the STRUCT or STRUCT item to receive the copy or copies.

source

is the STRUCT or STRUCT item from which the data is to be copied.

Result

#SETBYTES returns nothing.

Considerations

- If the entire source STRUCT or STRUCT item cannot be copied, #SETBYTES copies as many bytes as can fit.
- If the move changes the type definitions of the items, TACL might be unable to display or invoke the STRUCT.

#SETCONFIGURATION Built-In Function

#SETCONFIGURATION sets the TACL flags that can change the behavior of TACL for the specified TACL image or that can configure the currently running TACL process.

Note. These flags can be changed only by users who are authorized to alter the Command Interpreter Monitor (CMON). For additional information, see Command Interpreter Monitor Interface (CMON) on page 6-8.

```
#SETCONFIGURATION / option [, option ] .../
[ tacl-image-name ]
```

option

must be one or more of these:

AUTOLOGOFFDELAY *delay-value*

specifies the maximum number of minutes TACL is to wait at a prompt. If the specified time is exceeded, TACL automatically logs off, releases the default segment, and (if LOGOFFSCREENCLEAR is specified) clears the terminal screen. The default is -1 (disabled).

A TACL process issues a modem disconnect at autologoff.

BLINDLOGON { OFF | ON }

specifies the LOGON command, whether in the logged-off state or the logged-on state, prohibits the use of the comma, requiring the password to be entered at its own prompt while echoing is disabled. The default is OFF.

CMONREQUIRED [OFF | ON }

specifies that all operations requiring approval by $CMON be denied if $CMON is not available or is running too slowly. Approval of $CMON is not required if the TACL is already logged on as the super ID. The default is OFF.

△ **Caution.** If you are authorized to change CMONREQUIRED and intend to set it to ON, you must keep an unmodified copy of TACL for system operation use. Otherwise, you cannot log on if $CMON is not running or is running too slowly. If the modified TACL is in $SYSTEM.SYSnn, you cannot even start the system.

CMONTIMEOUT *timeout-value*

specifies the number of seconds that TACL is to wait for any $CMON operation. The default *timeout-value* is 30.

```
CONFIGRUN [ PROCESSCREATE | PROCESSLAUNCH ]
```

PROCESSLAUNCH

a TACL RUN[D] command is configured to call the system procedure PROCESS_LAUNCH_ to start a process. In this case, these additional RUN[D] command options, MAXMAINSTACKSIZE, MAXNATIVEHEAPSIZE, and GUARANTEEDSWAPSPACE, are available.

PROCESSCREATE

a TACL RUN[D] command is configured to call the system procedure PROCESS_CREATE_ to start a process. In this case, three additional RUN[D] command options, MAXMAINSTACKSIZE, MAXNATIVEHEAPSIZE, and GUARANTEEDSWAPSPACE, are not available. This is the default setting.

```
LOGOFFSCREENCLEAR { OFF | ON }
```

indicates if TACL is interactive and the IN file is a 65xx terminal, the terminal memory is cleared at (#)LOGOFF unless the NOCLEAR option is supplied.

The CLEAR and NOCLEAR options always override the automatic operation. The default is ON.

```
NAMELOGON { OFF | ON }
```

specifies the LOGON command, in both the logged-off and logged-on states. #CHANGEUSER built-in functions do not accept user numbers but rather require user names. The default is OFF.

```
NOCHANGEUSER { OFF | ON }
```

specifies the ability to log on from a logged-on state. The default is OFF.

```
REMOTECMONREQUIRED { OFF | ON }
```

specifies that all operations requiring approval by a remote $CMON be denied if the remote $CMON is unavailable or is running too slowly. $CMON approval is not needed if the TACL is already logged on as the super ID. The default is OFF.

```
REMOTECMONTIMEOUT remote-timeout-value.
```

specifies the number of seconds that TACL is to wait for any $CMON operation involving a remote $CMON. The default *remote-timeout-value* is 30.

```
REMOTESUPERID { OFF | ON }
```

specifies whether TACL can be started remotely by the super ID, or user ID 255, 255. The default is ON.

`REQUESTCMONUSERCONFIG [ OFF | ON ]`

if set to ON, then, after every LOGON command or #CHANGEUSER built-in function is executed, the TACL process sends a request to the $CMON process for the configuration parameters in effect for this current TACL user.

if set to OFF, then the configuration parameters in effect are always the TACL process default parameters, parameters obtained when a LOGOFF command has been executed and is followed (at some point) by a LOGON command, or parameters obtained when a noninteractive TACL process is started.

This option is initially set to OFF.

This option is meaningful only if the $CMON process has been coded to return the configuration information. Otherwise, the additional request is ignored. Use of this option allows $CMON to control (and change) the configuration parameters for each logon session: for example, for each user ID. These altered configuration parameters can then be returned to the TACL process.

Regardless of the setting, when a LOGOFF command has been executed and followed (at some point) by a LOGON command or when a noninteractive TACL process is started, the TACL process requests configuration information from the $CMON process.

`STOPONFEMODEMERR [ OFF | ON ]`

ON specifies that TACL stops when error 140 (FEMODEMERR) is encountered on its input. If the TACL process was started with the PORTTACL startup parameter, this TACL configuration setting is ignored. (TACL goes to the logged off state and waits for a modem connect message when error 140 is encountered.) The default is OFF, meaning TACL is put in a logged off state and waits for a modem connection message when an error 140 is encountered.

`tacl-image-name`

is the name of an existing TACL image file to be configured. You must that ensure a copy of TACL is created before trying to configure it, because TACL does not create the copy. If you omit *`tacl-image-name`*, the built-in function #SETCONFIGURATION configures the currently running TACL process and the new configuration values are used for later operations that require these configuration values.

Result

#SETCONFIGURATION returns zero if it configures the TACL image or itself successfully; otherwise, it returns these values:

-2	SETCONFIGURATION error	The specified image is not a TACL image.
-1	Insufficient capability error	The user does not have user ID 255, 255 .
>0	File system error	Number returned indicates the reason for the failure.

In addition, the built-in variable #ERRORNUMBERS is set when an error occurs while executing the #SETCONFIGURATION built-in function (there is no text output):

```
1165 error-detail-1 error-detail-2 0
```

The example following this description shows how to code #ERRORNUMBERS with #SETCONFIGURATION to determine the error. This table describes possible #ERRORNUMBERS values:

error-detail-1	*error-detail-2*	Cause	Effect	Recovery
1	*file-system error*	Depends on the error	Depends on the error	Depends on the error
2	0	The user ID is not 255,255.	Requested operation fails.	Logon using the super ID (255,255). Retry the operation.
3	1	Invalid TACL image file code is not 100.	Requested operation fails.	Retry the operation specifying a valid TACL image file.
3	2	None	None	Appears in the TACL code but is not implemented
3	3	Invalid TACL image file	Requested operation fails.	Retry the operation, specifying a valid TACL image file.
3	4	Invalid TACL image file (VPROC does not contain string “T9205”.)	Requested operation fails.	Retry the operation, specifying a valid TACL image file.
4	1	None	None	None
4	2	Invalid TACL image file	Requested operation fails.	Retry the operation, specifying a valid TACL image file.
5	0	Too many configuration options	Requested operation fails.	Retry the operation, specifying the correct number of configuration options.

Considerations

- If TACL cannot open or read the *tacl-image-file* or if it is not a TACL image, an error message results.
- A TACL image must be supplied for the #SETCONFIGURATION built-in function to configure the TACL. If the TACL image name is omitted, TACL will configure itself and the new configuration values are used for later operations that require these configuration values.
- TACL checks the specified image to ensure that it is a TACL image by examining the version procedure data space within the specified TACL image.
- Only user ID 255, 255 is allowed to modify the TACL configuration. If the user ID is not 255, 255, the #SETCONFIGURATION built-in function returns -1.
- If any option is not configurable, #SETCONFIGURATION terminates and leaves the image unmodified. This can occur with older TACL images.
- TACL configures only the specified option. Any options not specified remain unchanged.
- When starting a new TACL process, calls to the built-in function #SETCONFIGURATION from $SYSTEM.SYSTEM.TACLLOCL or TACLLOCL in the cold-loaded system, volume, and subvolume are executed regardless of the user ID. Placement of this built-in function and securing these TACLLOCL files is set by user ID 255, 255.
- If you specify the same option more than once, the last specification is the one used.
- The built-in variable #ERRORNUMBERS is set when an error occurs while executing the #SETCONFIGURATION.built-in function.
- If directly connected (no Safeguard or modem port control processes) to a NonStop system by starting a TACL process on a port (for example, a dial-in line or X.25 connection), the TACL being started on the port should be started with the PORTTACL startup parameter. TACL processes on all nodes that are accessible from the port should also have the TACL configuration parameter STOPONFEMODEMERR set to ON. This setting prevents a security breach situation caused by the error 140 (FEMODEMERR) processing by TACL. If all nodes do not have the TACL configuration parameter STOPONFEMODEMERR set to ON, the security breach situation still exists.

  The PORTTACL startup parameter and the STOPONFEMODEMERR TACL configuration parameter control the behavior of the local TACL process when the process receives an error 140 (FEMODEMERR) on its input. If TACL is not the foreground process, it is the responsibility of the foreground process to process the error 140 correctly. TACL cannot act on an error 140 unless it is the process that receives error 140.

If your installation uses $CMON processes and you want to take advantage of the STOPONFEMODEMERR behavior, your $CMON program must be changed to specifically include the new configuration parameter.

Example

```
?TACL MACRO
#FRAME

#PUSH configuration^error

==Make a duplicate TACL from the current TACL
FUP DUP $SYSTEM.SYSTEM.TACL, NEWTACL

==Secure the current TACL so no other users can access it.
FUP SECURE $SYSTEM.SYSTEM.TACL, "----"

==Configure the new TACL with the appropriate options

#SET configuration^error [#SETCONFIGURATION/BLINDLOGON OFF, &
CMONTIMEOUT 50/ NEWTACL]

[#IF configuration^error |THEN|

  == Pick up individual error fields:
  #PUSH tacl^error error^detail1 error^detail2
  #SETMANY tacl^error error^detail1 error^detail2,
      [#ERRORNUMBERS]
  [#CASE [error^detail1]

  |1| error [error^detail2] == File system error

  |2| #OUTPUT *ERROR* Not user 255, 255 == Security violation

  |3| #OUTPUT *ERROR* Incorrect TACL image == vproc error

  |4|  == Configuration error
      #OUTPUT *ERROR* Unable to fully configure TACL image

  |5| #OUTPUT *ERROR* Too many configuration options

  |OTHERWISE| #OUTPUT *ERROR* #SETCONFIGURATION
      == Unknown error

]

|ELSE| == No error in configuration

  ==Save the current TACL
  #RENAME $SYSTEM.SYSTEM.TACL, $SYSTEM.SYSTEM.OLDTACL

  ==Make the newly configured TACL the standard TACL
  #RENAME $SYSTEM.SYSTEM.NEWTACL, $SYSTEM.SYSTEM.TACL

]

#UNFRAME
```

#SETMANY Built-In Function

Use #SETMANY to distribute the members of a space-separated list into individual variable levels.

```
#SETMANY variable-name-list , [ text ]
```

variable-name-list

is a space-separated list of one or more existing variable levels or underscores.

text

is a space-separated list of items (if the entire function call is enclosed in square brackets, end-of-line characters are treated as spaces). Commas may occur in text because everything after the first comma is considered to be text. Space and end-of-line are the only characters considered to be separators in the text.

Result

#SETMANY returns nothing.

Considerations

- Each item in text is put into the corresponding variable level. If, however, the corresponding entry in *variable-name-list* is an underscore character (_) instead of a variable level name, the text item is discarded.
- If there are more items in text than there are entries in *variable-name-list*, the extra items are ignored. If there are more entries in *variable-name-list* than items in text, the unallocated variable levels are cleared.
- Specified variable levels must already exist; their types remain unchanged.
- To copy a string to an existing variable level, use the #SETV built-in function.
- To change the contents of a single variable level or built-in variable, use the #SET built-in function.

Examples

1. This example illustrates the use of #SETMANY to find out the four parts of a file name:

   ```
   [#SETMANY volume subvol file system,
     [#FILEINFO /VOLUME,SUBVOL,FILE,SYSTEM/ fname]
   ]
   ```

   The SYSTEM option is specified last because it returns nothing if the file name is in local form and would cause the correspondence between the two lists to be lost if it did not occur last.

2. This example shows how to find the numeric portion of the current operating system RVU:

   ```
   #SETMANY _ number , [#TOSVERSION]
   ```

   #TOSVERSION returns a letter, a space, and a number; the underscore causes the letter in the #TOSVERSION result to be discarded.

#SETPROCESSSTATE Built-In Function

Use #SETPROCESSSTATE to alter the value of a process state flag in the Process Control Block (PCB) of your logged-on TACL process.

The process state flags can only be set by the process owner.

```
#SETPROCESSSTATE / option / { 0 | 1 }
```

option

is one of these:

LOGGEDON

specifies the logged-on state of the TACL process. A value of 1 indicates that the TACL process is in a logged-on state and that the TACL process default file security, process access ID (PAID), creator accessor ID (CAID), and remote logon flags were set to their appropriate values by the operating system. The default value is 1.

TSNLOGON

specifies whether or not Safeguard software authenticated the TACL process. A value of 1 indicates that Safeguard software authenticated and created the TACL process or that the TACL process is a descendent of a local process that had the TSNLOGON flag set. The default value is 0 for a regular TACL process and 1 for a TSN-TACL process or a TACL process started by a TSN-TACL process.

When a descendent TACL process is created on a remote system, the TSNLOGON flag is set to 0 (by the operating system) for that remote process.

STOPONLOGOFF

specifies whether the current TACL process will be stopped after it enters a logged-off state. A value of 1 indicates that the current TACL process will be stopped when it enters a logged-off state. A value of 0 indicates that the current TACL process will not be stopped; instead, the TACL process will prompt for another logon. The default value is 0 for a regular TACL process or a TACL process started by a TSN-TACL process. The default is 1 for a TSN-TACL process.

If you set the STOPONLOGOFF flag to 1, the current TACL process stops when it enters a logged-off state.

PROPAGATELOGON

specifies how local descendent TACL processes start. A value of 1 for the current TACL process indicates that the INHERITEDLOGON state of the child TACL process will be 1. The new process will start in the logged-on state. A

value of 0 for the current TACL process indicates that the INHERITEDLOGON state of the local child TACL process will be 0. The new process will start in the logged-off state. The default value is 0.

Remote descendent TACL processes always start in a logged-off state.

You can clear the PROPAGATELOGON state flag in the current TACL process, which causes local descendent TACL processes to start in a logged-off state and prompt for logon. Remote descendent TACL processes always start in a logged-off state.

PROPAGATESTOPONLOGOFF

specifies how local child TACL processes stop. A value of 1 for the current TACL process indicates that the STOPONLOGOFF state of the child TACL process will be 1. The child process will stop when it enters a logged-off state. A value of 0 for the current TACL process indicates that the STOPONLOGOFF state of a local child process will be 0. The child process will prompt for logon information when it enters a logged-off state. The default value is 0.

If you set the PROPAGATESTOPONLOGOFF flag to 1, local descendent TACL processes are stopped when they enter a logged-off state.

For remote child TACL processes, the parent process propagates a value of 0 for the STOPONLOGOFF flag.

Results

The #SETPROCESSSTATE built-in function returns a zero, indicating success, or a positive number indicating a file system error.

If the user does not have sufficient privilege (as in a security violation), TACL returns a file system error 48. This error indicates that the user cannot alter the state, although the user may be the owner of the TACL process.

Considerations

- The term TSN-TACL refers to a TACL process that Safeguard software starts after authenticating the user of the TACL process.
- For security reasons, most process state flags cannot be altered by non-privileged users. Table 9-13 on page 9-357 describes valid operations:

Table 9-13. Valid Operations for #SETPROCESSSTATE Built-In Function

Flag	Regular TACL Process	TSN-TACL Process	TACL Process Started By a TSN-TACL Process
LOGGEDON	N/A	Can be set but not cleared	N/A
STOPONLOGOFF	Can be set and cleared	Can be set and cleared	Can be set and cleared
PROPAGATELOGON	Can be cleared but not set	Can be cleared but not set	Can be cleared but not set
PROPAGATESTOPONLOGOFF	Can be set and cleared	Can be set and cleared	Can be set and cleared
TSNLOGON	Cannot be set or cleared	Cannot be set or cleared	Cannot be set or cleared

- To retrieve process state information for a TACL process, use the #GETPROCESSSTATE built-in function. For more information about process state data, see the PROCESS_GETINFOLIST_ and PROCESS_SETINFO_ procedures in the *Guardian Procedure Calls Reference Manual*.

#SETSCAN Built-In Function

Use #SETSCAN to specify the position at which the next #ARGUMENT function is to resume processing arguments.

```
#SETSCAN num
```

num

specifies the argument position. Position zero is the first character after the routine name.

Result

#SETSCAN returns nothing.

Considerations

- To obtain the number of characters that #ARGUMENT has processed, use the #GETSCAN built-in function.
- To determine whether an entire argument set has been processed, use the #MORE built-in function.

#SETSYSTEMCLOCK Built-In Function (Super-Group Only)

Use #SETSYSTEMCLOCK to change the setting of the system clock. To use this function, you must have a group ID of 255.

```
#SETSYSTEMCLOCK julian-gmt mode [ tuid ]
```

julian-gmt

is the current Julian timestamp.

mode

is a number that specifies the mode and source, as follows:

Number	Mode	Source
0	Absolute Greenwich mean time	Operator input
1	Absolute GMT	Hardware clock
2	Relative GMT	Operator input
3	Relative GMT	Hardware clock
5	Relative GMT	Force set
6	Relative GMT	Force adjustment
7	Absolute GMT	Force set
8	Ignored, optional	Stop time adjustment
9	Rate adjustment	Adjust rate
10	Ignored, optional	Clear rate adjustment

A relative mode implies that the *julian-gmt parameter* contains a microsecond-based time adjustment, not a timestamp. This mode is used for precise time synchronization to a hardware clock or for a moderately precise method for the operator to adjust the time. For details about these modes, see the description of procedure SYSTEMCLOCK_SET_ in the *Guardian Procedure Calls Reference Manual*.

Modes 9 and 10 are valid only if the system library contains the SYSTEMCLOCK_SET_ Guardian procedure.

tuid

is a time update ID obtained from #JULIANTIMESTAMP; use it with modes 2 and 3 to avoid conflicting changes.

Result

#SETSYSTEMCLOCK returns zero if it sets the system clock successfully; otherwise, it returns a negative error result. If the system library contains the SYSTEMCLOCK_SET_ Guardian procedure, #SETSYSTEMCLOCK returns the error result defined in the "Result Codes (Error Returns)" subsection of the

SYSTEMCLOCK_SET_ description in the *Guardian Procedure Calls Reference Manual*. If not, the error result is -1.

Considerations

- The #SETSYSTEMCLOCK built-in function calls the SETSYSTEMCLOCK system procedure.
- If you use #SETSYSTEMCLOCK to set the system clock forward or backward two minutes or less, the system adjusts the clock in small increments rather than setting it to the new time. Adjusting the clock forward two minutes takes about 33 hours. Adjusting the clock back two minutes takes about 14 days.

  If you issue two SETSYSTEMCLOCK commands in less than ten seconds, the system stops any ongoing adjustment and sets the clock to the second value.

Example

This example calls #SETSYSTEMCLOCK and returns an error if you do not have super-group capability:

```
#FRAME
#PUSH tstamp
[#IF [#SETSYSTEMCLOCK tstamp 0] |THEN|
#OUTPUT System time not changed. Insufficient capability.
]
#UNFRAME
```

#SETV Built-In Function Use

#SETV to copy a string to an existing variable level.

```
#SETV dest-variable-level source-variable
```

dest-variable-level

is the variable level that is to receive the copy; it must already exist. Its original contents are lost.

source-variable

is the name of an existing variable that contains the string to be copied. *source-string* remains unchanged. It must not be of type DIRECTORY, be text enclosed in quotation marks, or be a concatenation of such entities. The concatenation operator is '+' (the apostrophes are required).

Result

#SETV returns nothing.

Considerations

- If the destination variable is of a different type than the source variable, the destination variable inherits the type as well as the contents of the source variable.
- If a variable maintains a connection to a file, device, or process, such as a connection established by #REQUESTER, and #SETV sets the variable to a new value, the original connection is dropped because TACL deletes the original contents of the variable.
- When the source string is a structure, the destination variable level becomes a structure like the source structure and contains its own copy of the data.
- To change the contents of a variable level or built-in variable, use the #SET built-in function.
- To distribute the members of a space-separated list into several individual variable levels, use the #SETMANY built-in function.

Example

This example shows a combination of a variable name and quoted text in the source string:

```
#PUSH var termname
#SET termname [#MYTERM]
#SETV var "My terminal is " '+' termname '+' " at this time."
```

Assuming that the home terminal is named $HAPPY at the time the #SET function is invoked, var then contains:

```
My terminal is $HAPPY at this time.
```

#SHIFTDEFAULT Built-In Variable

Use #SHIFTDEFAULT to set or obtain the current default shift direction for #SHIFTSTRING.

```
#SHIFTDEFAULT
```

Result

#SHIFTDEFAULT returns DOWN, NOOP, or UP.

Considerations

- When you first log on, #SHIFTDEFAULT is initialized to UP.
- Use #PUSH #SHIFTDEFAULT (or PUSH #SHIFTDEFAULT) to save a copy of your current default shift setting.
- Use #POP #SHIFTDEFAULT (or POP #SHIFTDEFAULT) to restore the default shift setting from the last copy pushed.
- Use #SET #SHIFTDEFAULT (or SET VARIABLE #SHIFTDEFAULT) to set the default shift direction (DOWN, NOOP, or UP) of shifting by #SHIFTSTRING.

  The syntax of #SET #SHIFTDEFAULT is:

  ```
  #SET #SHIFTDEFAULT { DOWN | NOOP | UP }
  ```

  `DOWN`

  causes #SHIFTSTRINGs with no UP or DOWN option to shift to lowercase.

  `NOOP`

  causes #SHIFTSTRINGs with no UP or DOWN option to do no shifting.

  `UP`

  causes #SHIFTSTRINGs with no UP or DOWN option to shift to uppercase.

#SHIFTSTRING Built-In Function

Use #SHIFTSTRING to change text from uppercase to lowercase and from lowercase to uppercase.

```
#SHIFTSTRING [ / option / ] [ text ]
```

option

specifies how the text should be shifted. If you omit the option, the shift direction defaults to the current value of the #SHIFTDEFAULT built-in variable. *option* is one of these:

DOWN

specifies that the text is to be shifted to lowercase.

NOOP

specifies that the text is not to be shifted.

UP

specifies that the text is to be shifted to uppercase.

text

is the text to be shifted. If not specified, #SHIFTSTRING does nothing.

Result

#SHIFTSTRING returns the text argument, shifted as specified.

Considerations

- If you do not specify a shift direction, it defaults to the current value-DOWN, NOOP, or UP-of the #SHIFTDEFAULT built-in variable.
- If you specify the NOOP option, or if the current #SHIFTDEFAULT setting is NOOP, #SHIFTSTRING does nothing.
- If the CPRULES0 character-processing rules are in effect, a number of lowercase characters (but not all) in the upper half of the international character set lose their diacritical marks when upshifted. Subsequent downshifting does not restore them. For example:

Upshift	Downshift
ãiú = ÃIU	ÃIU = ãiu

  When the CPRULES1 rules are in effect, diacritical marks remain intact when characters that have them are upshifted.

Example

This example illustrates the use of #SHIFTSTRING to shift the case of a variable level:

```
62> #PUSH vara
63> #SET vara this is a test
64> #OUTPUT [#SHIFTSTRING /UP/ [vara]]
THIS IS A TEST
```

#SORT Built-In Function

Use #SORT to sort a space-separated list of text.

```
#SORT [ / option / ] [ text ]
```

option

specifies the order in which the indicated text is to be sorted. If you omit option, ascending order is the default. If, however, there is any chance that the text may begin with a slash (/), you must include an option so that the slash in the text is not interpreted as the beginning of the option. *option* is one of these:

ASCENDING

specifies that the text is to be sorted in ascending order according to the ASCII collating sequence.

DESCENDING

specifies that the text is to be sorted in descending order according to the ASCII collating sequence.

text

specifies a space-separated list to be sorted. If not specified, #SORT does nothing.

Result

#SORT returns a sorted space-separated list.

Considerations

- #SORT upshifts all characters before comparing them, so uppercase and lowercase forms of the same character sort as equal; equal characters do not change position with regard to each other during sorting.
- When dealing with characters from the upper half of the international character set, the character-processing rules in effect have an influence on #SORT (see Examples).

Examples

1. This example illustrates the use of #SORT on a specified list of characters:

```
65> [#DEF pogo TEXT |BODY| albert howland churchy
grundoon]
66> #OUTPUT [#SORT /DESCENDING/ [pogo]]
albert churchy grundon howland
```

2. This example illustrates the way in which equal-valued uppercase and lowercase characters are sorted.

```
67> #OUTPUT [#SORT /ASCENDING/ a B b A]
a A B b
```

3. These examples show the influence of character-processing rules. The first instance shows the result when the CPRULES0 rules, which cause a number of international characters to lose their diacritical marks when upshifted, are in effect:

```
68> #OUTPUT [#SORT /ASCENDING/ B a a A b]
a â A B b
```

   The second instance shows the result when CPRULES1 rules, under which diacritical marks remain intact during upshifting, are in effect:

```
69> #OUTPUT [#SORT /ASCENDING/ B a â A b]
a A B b â
```

#SPIFORMATCLOSE Built-In Function

Use #SPIFORMATCLOSE to close an open EMS formatter template file.

```
#SPIFORMATCLOSE
```

Result

#SPIFORMATCLOSE returns nothing.

Consideration

TACL opens the formatter template file associated with =_EMS_TEMPLATES automatically whenever you invoke a built-in function that uses a formatter file (such as #EMSTEXT or #EMSINIT). You can use the #SPIFORMATCLOSE built-in function to close the EMS formatter template file so that you can change the =_EMS_TEMPLATES DEFINE and open a new template file.

For more information, see the *DSM Template Services Manual*.

#SSGET Built-In Function

Use #SSGET to retrieve binary token values from an SPI buffer, convert them to external representation, and make that external representation accessible in the function result.

You cannot use #SSGET to extract values of extensible structured tokens using a token map or using a token code of type ZSPI^TDT^STRUCT. Use #SSGETV instead.

```
#SSGET [ / option [ , option ] ... / ] buffer-var get-op
```

option

is any of these:

COUNT *count*

is the maximum number of token values to be returned. This option specifies that the token value returned is an array of count elements, each of which is described by *token-code*. If this option is omitted, it defaults to 1.

If count is greater than 1, #SSGET continues searching until it either satisfies the requested count or reaches the end of the buffer or list.

If count is less than 1, an error occurs.

INDEX *index*

is the specific occurrence of *token-code*, starting from the beginning of the buffer or list (an index of 1 gets the first occurrence of that token code, 2 the second, and so on). If you omit this option or if index = 0, #SSGET returns the next occurrence of the token code after the current position and resets the current position to that of the token value returned.

If you want to search from the beginning of the buffer, you must supply a nonzero index or else have previously reset the initial position with #SSPUT(V) using ZSPI^TKN^INITIAL^POSITION or ZSPI^TKN^RESET^BUFFER.

If index is less than zero, an error occurs.

SSID *ssid*

is a subsystem ID that qualifies the token code; if it is omitted or zero (0.0.0), it defaults to the subsystem ID of the current list or, if the current position is not in a list, to the subsystem ID specified in the SPI message header. The version field of this parameter is not used when searching the buffer.

buffer-var

is the name of the message buffer from which information is to be taken; it must be a writable STRUCT that has been initialized by #SSINIT.

get-op

is one of these:

token-code

directs #SSGET to return the token value or values associated with *token-code*. If *token-code* is a token that marks the beginning of a list, #SSGET selects the list so that subsequent calls can retrieve tokens within the list. If *token-code* is ZSPI^TKN^ENDLIST, #SSGET pops out of the list.

token-code can be any of the header tokens described for the #SSGET Built-In Function on page 9-369 and #SSGETV Built-In Function on page 9-374. You can also supply the token ZSPI^TKN^DEFAULT^SSID to obtain the default subsystem ID at the current position.

ZSPI^TKN^COUNT *c-token-id*

directs #SSGET to return the number of occurrences of the token specified by the token code or the token map *c-token-id*, starting with the occurrence specified by index. To count all occurrences in the current list, specify an index of 1.

If *c-token-id* is omitted or equal to ZSPI^VAL^NULL^TOKENCODE, and index is omitted or zero, #SSGET counts occurrences of the current token beginning with the current occurrence.

ZSPI^TKN^LEN *l-token-id*

directs #SSGET to return the byte length of the token specified by the token code or token map *l-token-id*. This is the size of the buffer needed to contain the stated occurrence of the token value. For variable-length token values, this includes the two bytes required for the length word: The byte length returned is *token-value*[0]+2.

If you omit this option, or if *l-token-id* is equal to ZSPI^VAL^NULL^TOKENCODE, and index is omitted or zero, #SSGET returns the length of the current occurrence of the current token.

If *l-token-id* is a token map, this operation returns the length contained in that map; the actual value in the buffer can differ from this length. To get the actual length of the token value in the buffer, invoke #SSGET with ZSPI^TKN^LEN and a token code made up of ZSPI^TYP^STRUCT and the token number from the token map. This invocation returns the length of the structure value, including two bytes for the length field. Subtract 2 from this value to get the length of the value itself.

ZSPI^TKN^NEXTCODE

directs #SSGET to return the next token code that is different from the current token code, followed by the subsystem ID. The subsystem ID returned in the result always has a version field of zero (null).

The index parameter has no effect on this operation, but if you supply it, it must be zero.

ZSPI^TKN^NEXTTOKEN

directs #SSGET to return the very next token code, followed by the subsystem ID. The subsystem ID returned always has a version field of zero (null).

This operation differs from ZSPI^TKN^NEXTCODE in that it always returns the token code of the next token, whether it is the same as that of the current token or different, and whether the token is within a list or not. The operation returns multiple occurrences of the same token code in the same order as they were added to the buffer with #SSPUT(V).

The index and count parameters have no effect on this operation, but if you use index, it must be zero; count is always returned as 1 in this operation.

Note. The special operations ZSPI^TKN^NEXTCODE and ZSPI^TKN^NEXTTOKEN return only token codes. In particular, note that tokens added to the buffer using #SSPUTV with a token map are carried in the buffer with a token code of type ZSPI^TYP^STRUCT. The NEXTCODE and NEXTTOKEN operations return that token code, not the token map used with #SSPUTV.

ZSPI^TKN^OFFSET *o-token-id*

directs #SSGET to return the byte offset of the token specified by the token code or token map *o-token-id*. The value returned is the offset from the start of the buffer to the value associated with the specified token code and index. (For variable-length values, the token value begins with the length word; the offset given is the offset to that length word.)

If you omit this option or if *o-token-id* is equal to ZSPI^VAL^NULL^TOKENCODE, and index is omitted or zero, #SSGET returns the length of the current occurrence of the current token.

Note that you must supply appropriate token code definitions. TACL merely keys off the numeric token codes for the special operations.

TACL supports the special semantics for only those SPI special token codes shown; any other token codes are assumed to adhere to standard semantics.

Result

#SSGET returns a numeric status code indicating the outcome of the SSGET procedure. If the status code is zero (no error), it is followed by a space and a space-separated list of the relevant SSGET results in the TACL external representation, as follows:

- If you specify *token-code*, the number of token values returned, followed by a space-separated list of those values in external form; variable-length token values

are returned in two parts-the byte length followed by the actual value-separated by a space

- If you specify ZSPI^TKN^COUNT *c-token-id*, the total number of occurrences of the specified token, starting at the occurrence specified by index If you specify ZSPI^TKN^LEN *l-token-id*, the length of the token specified by the token code or token map *l-token-id*
- If you specify ZSPI^TKN^NEXTCODE, the next token code that is different from the current token code, followed by the subsystem ID
- If you specify ZSPI^TKN^NEXTTOKEN, the next token code, regardless of whether it is different from the current token code, followed by the subsystem ID
- If you specify ZSPI^TKN^OFFSET *o-token-id*, the byte offset of the token specified by token code or token map *o-token-id*

If the status code is not zero, its meaning is:

Code	Condition
-1	Invalid buffer format
-2	Invalid parameter value
-3	Missing parameter
-4	Invalid parameter address
-5	Buffer full
-6	Invalid checksum
-7	Internal error
-8	Token not found
-9	Invalid token code or map
-10	Invalid subsystem ID
-11	Operation not supported
-12	Insufficient stack space

Considerations

- Tokens extracted by #SSGET are not deleted or removed from the buffer.
- When the current position is within a particular list, all #SSGET calls pertain only to tokens within that list (except that header fields are always accessible). You can exit from the list by calling #SSGET to get the ZSPI^TKN^ENDLIST token.
- When *token-code* is ZSPI^TKN^ENDLIST, the index and count parameters have no effect. However, if you supply them, index must be 0 or 1; count is always returned as 1.

Header Tokens and Special Operation for #SSGET and #SSGETV

Table 9-14 lists the header tokens used to obtain the values of various fields of the SPI message header. The index parameter you specify with these token codes must be 0 or 1.

Table 9-14. #SSGET(V) Header Tokens

Token Code	Type	Item Retrieved
ZSPI^TKN^CHECKSUM	INT	Checksum flag
ZSPI^TKN^COMMAND	ENUM	Command number
ZSPI^TKN^HDRTYPE	UINT	Header type
ZSPI^TKN^LASTERR	ENUM	Last nonzero SPI status code
ZSPI^TKN^LASTERRCODE	INT2	*token-code*, *c-token-id*, or *l-token-id* on last error call
ZSPI^TKN^LASTPOSITION	BYTE:8	Position of last token added with SSPUT
ZSPI^TKN^MAX^FIELD^VERSION	UINT	Maximum field version
ZSPI^TKN^MAXRESP	INT	Maximum response records to return
ZSPI^TKN^OBJECT^TYPE	ENUM	Object-type number
ZSPI^TKN^POSITION	BYTE:8	Current position for #SSGET
ZSPI^TKN^SERVER^VERSION	UINT	Server RVU
ZSPI^TKN^SSID	SSID	Subsystem ID used with #SSINIT
ZSPI^TKN^USEDLEN	UINT	Number of bytes used in the buffer

If you specify ZSPI^TKN^LASTPOSITION or ZSPI^TKN^POSITION, the value returned is an eight-byte position descriptor that you can later use to reset the position with the #SSPUT or #SSPUTV special operation ZSPI^TKN^POSITION. For #SSGET, the position descriptor is returned as a space-separated list of eight one-byte values. For #SSGETV, the value is returned in a STRUCT consisting of eight bytes.

You can also use the special token code ZSPI^TKN^DEFAULT^SSID to obtain the default subsystem ID at the current position, preceded by the number of token values returned (which in this case is always 1). #SSGET(V) and #SSPUT(V) use this value whenever the SSID parameter is omitted or null.

If the default subsystem ID comes from the owner of a list, the version field of SSID is set to ZSPI^VAL^NULL^VERSION. Your function should therefore omit the version field when comparing subsystem ID values for equality.

#SSGETV Built-In Function

Use #SSGETV to obtain binary token values from an SPI buffer and put them into a STRUCT. You can use #SSGETV with any type of token (you must use #SSGETV with tokens of type ZSPI^TYP^STRUCT and extensible structured tokens).

```
#SSGETV [ / option [ , option ] ... / ] buffer-var get-op
  result-var
```

option

is any of these:

```
COUNT count
INDEX index
SSID ssid
```

These options are the same as those described under #SSGET, substituting *token-id* for all references to *token-code*.

buffer-var

is the same as that described for #SSGET.

get-op

is one of these:

token-id

is either a token code or a token map. It directs #SSGETV to return the token value or values associated with *token-id*.

If *token-id* is a token that marks the beginning of a list, #SSGETV selects the list so that subsequent calls can retrieve tokens within the list. If *token-code* is ZSPI^TKN^ENDLIST, #SSGETV pops out of the list.

```
ZSPI^TKN^COUNT c-token-id
ZSPI^TKN^LEN l-token-id
ZSPI^TKN^NEXTCODE
ZSPI^TKN^NEXTTOKEN
ZSPI^TKN^OFFSET o-token-id
```

are the same as those in #SSGET, but #SSGETV returns results in *result-var* instead of in the function result.

Note. You must supply appropriate token code definitions. TACL supports the special semantics for only those SPI special token codes shown. Any other token codes are assumed to adhere to standard semantics.

---

result-var

is the name of the writable STRUCT into which #SSGETV is to store the data returned. The original contents of the STRUCT are lost.

If the status code in the function result is zero (no error), the result stored in *result-var* is as follows:

- If you specified *token-id*, the result is the value of the token.
- If you specified ZSPI^TKN^COUNT *c-token-id*, the result is an INT giving the number of occurrences of the token specified by *c-token-id*, starting at index.
- If you specified ZSPI^TKN^LEN *l-token-id*, the result is an INT giving the length of the specified token.
- If you specified ZSPI^TKN^NEXTCODE, the result is an INT2 giving the next token code that is different from the current token code, an INT giving the number of contiguous occurrences of that token code, and the subsystem ID.
- If you specified ZSPI^TKN^NEXTTOKEN, the result is an INT2 giving the next token code, whether different from or identical to the current token code, followed by the subsystem ID.
- If you specified ZSPI^TKN^OFFSET *o-token-id*, the result is an INT2 giving the byte offset of the token specified by *o-token-id*.

Result

#SSGETV returns a numeric status code indicating the outcome of the SSGET procedure. The meaning of the status code is as follows:

Code	Condition
0	No error
-1	Invalid buffer format
-2	Invalid parameter value
-3	Missing parameter
-4	Invalid parameter address
-5	Buffer full
-6	Invalid checksum
-7	Internal error
-8	Token not found
-9	Invalid token code or map
-10	Invalid subsystem ID

If no error occurred, and if *get-op* is *token-id*, the status code is followed by a space and the number of token values returned.

Considerations

- Tokens extracted by #SSGETV are not deleted or removed from the buffer.
- When the current position is within a particular list, all #SSGETV calls pertain only to tokens within that list (except that header fields are always accessible). You can exit from the list by calling #SSGETV to get the ZSPI^TKN^ENDLIST token.
- When *token-id* is ZSPI^TKN^ENDLIST, the index and count parameters have no effect. However, if you supply them, index must be 0 or 1; count is always returned as 1.
- When using #SSGETV with a token map for the *token-id* parameter, the map can specify a structure version that is longer or shorter than the structure contained in the buffer. If the requested version is longer than the version in the buffer, #SSGETV calls SSNULL to set to null values the new fields that are not obtained from the buffer. If the requested version is shorter than the one in the buffer, #SSGETV returns only the requested length.
- If the data returned by #SSGETV is longer than the data area of the STRUCT identified by *result-var*, TACL discards the excess bytes without notification. If the data is shorter than the data area of *result-var*, TACL sets the entire STRUCT to its default values, then overwrites the beginning of the data bytes of the STRUCT with the returned data. No type conversions of any kind are done. Consequently, if the token retrieved is of type ZSPI^TYP^INT, for instance, and the *result-var* STRUCT consists of a single field of type INT2, the token value appears in the high-order 16 bits of the INT2 field, not the low-order 16 bits.
- If you specified the COUNT option, TACL puts all occurrences of the token value into the STRUCT exactly as returned by #SSGETV, subject to the size constraints mentioned in the previous consideration. If the tokens are variable-length tokens, each token value consists of a length word followed by the actual value, and the actual value is word-aligned.
- You can pass header tokens and one special operation in *token-id* to get corresponding values. They are described under “Header Tokens and Special Operation for #SSGET and #SSGETV” in the explanation of #SSGET.

Examples

1. This example shows how to declare STRUCTs that allow you to extract individual fields of the token code or the subsystem ID returned by #SSGETV with the ZSPI^TKN^NEXTCODE or ZSPI^TKN^NEXTTOKEN option:

```
?SECTION decompose_ssid STRUCT
  BEGIN
    SSID ss;
    STRUCT z^ssid REDEFINES ss;
      BEGIN
        CHAR z^owner(0:7);
        INT z^number;
```

```
        UINT z^version;
      END;
  END;

?SECTION nexttoken_return STRUCT
  BEGIN
    STRUCT tkn ; LIKE zspi^ddl^tokencode;
    STRUCT ssid; LIKE decompose_ssid;
  END;

?SECTION nextcode_return STRUCT
  BEGIN
    STRUCT tkn ; LIKE zspi^ddl^tokencode;
    INT contiguous_occurrences;
    STRUCT ssid; LIKE decompose_ssid;
  END;
```

2. This routine uses these STRUCT declarations to compare two subsystem IDs returned by #SSGETV with ZSPI^TKN^NEXTCODE or ZSPI^TKN^NEXTTOKEN, ignoring the version field:

```
?SECTION same_ssid ROUTINE == <ssid1> <ssid2>
== Returns TRUE if two SSIDs are the same except for
== the version field
#FRAME
#PUSH sstext
#DEF ss1 STRUCT LIKE decompose_ssid;
#DEF ss2 STRUCT LIKE decompose_ssid;
#IF{SINK} [#ARGUMENT/VALUE sstext/ SUBSYSTEM]
#SET ss1 [sstext]
#IF{SINK} [#ARGUMENT/VALUE sstext/ SUBSYSTEM]
#SET ss2 [sstext]
#RESULT [#COMPUTE [#COMPAREV ss1:z^ssid:z^owner(0:7)
                             ss2:z^ssid:z^owner(0:7)]
              AND [#COMPAREV ss1:z^ssid:z^number
                             ss2:z^ssid:z^number] ]
#UNFRAME
{same_ssid}
```

#SSINIT Built-In Function

Use #SSINIT to initialize a STRUCT as an SPI message buffer, preparing it for use with the other #SSxxx built-in functions. #SSINIT gives the buffer an appropriate header and, optionally, adds parameter information.

Use #SSINIT only to initialize a buffer for a command or a response; do not use it to initialize an event-message buffer.

```
#SSINIT [ / TYPE 0 / ] buffer-var ssid command
  [ / type-0-option [ , type-0-option ] ... / ]
```

`TYPE 0`

declares the header type of the SPI message buffer being initialized. Type 0, a command or response header, is the default. Type 0 is the only type supported by TACL at this time.

`buffer-var`

is the name of a writable STRUCT to be used as an SPI buffer. #SSINIT automatically passes to the SSINIT procedure the data length of the STRUCT. Any current contents of the STRUCT are lost.

`ssid`

is the subsystem ID of the subsystem. For requester functions, this subsystem ID identifies the target subsystem; the version field must identify the version of the subsystem definitions that your requester function is using. For server functions (subsystems), this subsystem ID must identify your server, including its version.

`command`

is the command number.

`type-0-option`

can be any of these:

`CHECKSUM` *`num`*

is the checksum flag. If num is zero, checksum protection of the data portion of the buffer is disabled; if it is a nonzero value, checksum protection of the data portion is enabled. If you omit this option, it defaults to zero.

`MAXRESPONSES` *`num`*

is the maximum number of response records to be returned by the subsystem in each reply message. A value of zero specifies one response record per reply, not enclosed in a list. Any value greater than zero specifies up to that many response records, each enclosed in a list. A value of -1 specifies as

many response records as can fit in the buffer, each enclosed in a list. If you omit this option, it defaults to zero.

OBJECT *num*

is the object type. If you omit this option, it defaults to ZSPI^VAL^NULL^OBJECT^TYPE (zero).

SERVERVERSION *num*

is the server version number, normally provided only by subsystems or by other programs or functions that act as servers. This number is a 16-bit unsigned integer value representing the RVU of the subsystem or server. SSINIT puts this value in the header token ZSPI^TKN^SERVER^VERSION for use in RVU compatibility checking. If you omit this option, the server RVU defaults to zero.

Result

#SSINIT returns a numeric status code indicating the outcome of the SSINIT procedure, as follows:

Code	Condition
0	No error
-2	Invalid parameter value
-3	Missing parameter
-4	Invalid parameter address
-5	Buffer full
-7	Internal error
-10	Invalid subsystem ID
-12	Insufficient stack space

Consideration

For the SSID parameter, you generally use the subsystem ID defined by the target subsystem. For most subsystems, the subsystem ID has a name of the form *subsys*^VAL^SSID.

Example

Here is an example using the TMF subsystem:

```
#SSINIT buf [ZTMF^VAL^SSID] [ZSPI^VAL^GETVERSION]
```

#SSMOVE Built-In Function

Use #SSMOVE to copy tokens from one message buffer to another. #SSMOVE performs a sequence of #SSGETV and #SSPUTV operations.

```
#SSMOVE [ / option [ , option ] ... / ]
  source-var dest-var token-id
```

option

can be any of these:

COUNT *count*

specifies the maximum number of token values to move, unless *token-id* is a list token; in that case, count specifies the maximum number of lists to move. If you omit this option, count defaults to 1.

DINDEX *dest-index*

if *dest-index* is greater than zero, it identifies the first occurrence of *token-id* to be replaced in the destination buffer. A value of 1 specifies that replacement is to start with the first occurrence of the token code, 2 specifies the second occurrence, and so on. If the specified occurrences are not found in the destination buffer, #SSMOVE appends the tokens being moved to the end of the buffer.

If *dest-index* is zero, it directs #SSMOVE to add the tokens from the source buffer to the end of the destination buffer. If you omit this option, *dest-index* defaults to zero.

SINDEX *source-index*

if *source-index* is greater than zero, it identifies the first occurrence of *token-id* to be copied from the source buffer. One occurrence or multiple occurrences can be moved, depending on the value of the count parameter. A *source-index* value of 1 specifies that copying is to start with the first occurrence of the token code, 2 specifies the second occurrence, and so on.

If *source-index* is zero, #SSMOVE selects the next occurrence of the token code after the current position in the source buffer. If you omit this option, *source-index* defaults to zero.

SSID *ssid*

is a subsystem ID that qualifies the token ID. If you omit it, or if it is equal to zero (0.0.0), it defaults to the subsystem ID of the current list or, if the current position is not in a list, to the subsystem ID specified in the SPI message header (ZSPI^TKN^SSID). The version field of SSID is not used in searching the source buffer.

source-var

is the name of the source message buffer variable level, from which the specified token or tokens are to be copied. *source-var* must be a writable STRUCT that has been initialized with #SSINIT.

dest-var

is the name of the destination message buffer variable level, to which the specified token or tokens are to be copied. *dest-var* must be a writable STRUCT that has been initialized with #SSINIT.

token-id

is a token code or a token map that identifies the token to be moved. That token must be present in the source buffer. The token ID can refer to a simple token, an extensible structured token, or a list token. If *token-id* identifies a list token, #SSMOVE moves that token, its associated end-list token, and all tokens in between.

Result

#SSMOVE returns a numeric status code indicating the outcome of the SSMOVE procedure, as follows:

Code	Condition
0	No error
-1	Invalid buffer format
-2	Invalid parameter value
-3	Missing parameter
-4	Invalid parameter address
-5	Buffer full
-6	Invalid checksum
-7	Internal error
-8	Token not found
-9	Invalid token code or map
-10	Invalid subsystem ID
-11	Operation not supported
-12	Insufficient stack space

If the status code is zero (no error), it is followed by a space and the count of the token values or lists moved.

Considerations

- #SSMOVE does not alter any tokens copied from the source buffer.
- After a successful copy, #SSMOVE changes the source buffer position for a subsequent #SSGET(V) to the position of the last token moved.
- When #SSMOVE copies a token identified by a token map, it truncates or pads the value according to the map specifications, and adjusts the ZSPI^TKN^MAX^FIELD^VERSION header field of the destination buffer appropriately.
- You can use #SSMOVE to move an incomplete list (one with no end-list token) if, and only if, you omit the DINDEX option or specify *dest-index* as zero. If you supply a nonzero *dest-index*, meaning that you are requesting a replacement operation, an incomplete list causes #SSMOVE to return a “token not found” status code (-8).
- If an error occurs on #SSMOVE, you can set the ZSPI^TKN^LASTERR and ZSPI^TKN^LASTERRCODE indications in either the source buffer or the destination buffer, depending on whether the error occurred on the logical #SSGETV or #SSPUTV part of the copy.

#SSNULL Built-In Function

Use #SSNULL to initialize an extensible structured token before setting values within the structure.

Note. A macro or routine must use #SSNULL before you place values in the fields of an extensible structured token, even if all currently defined fields are to be set explicitly. Using #SSNULL allows the routine to continue to work with future software RVUs.

```
#SSNULL token-map struct
```

token-map

is a token map to be used in initializing the fields of the structure.

struct

is the STRUCT to be initialized with null values.

Result

#SSNULL returns a numeric status code indicating the outcome of the SSNULL procedure, as follows:

Code	Condition
0	No error
-3	Missing parameter
-4	Invalid parameter address
-7	Internal error
-9	Invalid token code or map

#SSPUT Built-In Function

Use #SSPUT to convert token values from external representation to binary form and put them into an SPI buffer previously initialized by #SSINIT.

You cannot use #SSPUT to insert values of extensible structured tokens using a token map or using a token code of type ZSPI^TDT^STRUCT. Use #SSPUTV instead.

```
#SSPUT [ / option [ , option ] ... / ] buffer-var
  token-code [ token-value [ token-value ] ... ]
```

option

is either of these:

COUNT *count*

is an integer in the range 1 through 65535 that specifies the token count; you must supply that number of token values. If count is greater than 1, it states that *token-value* is a space-separated list of count elements, each of which is described by *token-id*. (Note that you must state a variable-length value in two parts-the byte length, followed by the actual value-separated by a space.) If you omit COUNT, you must supply exactly one value (the default).

If *token-code* denotes a special operation whose semantics do not allow a token value, you must omit COUNT. For the special token code ZSPI^TKN^DELETE, you must supply a count value; #SSPUT interprets it as the index value of the token code to be deleted.

SSID *ssid*

specifies the subsystem ID that qualifies the token code. If you omit this option or if SSID is zero (0.0.0), *ssid* defaults to the subsystem ID of the current list or, if the current position is not in a list, to the subsystem ID specified in the SPI message header (ZSPI^TKN^SSID).

buffer-var

is the name of the SPI message-buffer variable level into which tokens are to be placed; the variable level must be a writable STRUCT that has been initialized by #SSINIT.

token-code

either identifies the token whose token value is being supplied or denotes a special operation (described under “Header Tokens and Special Operations for #SSPUT and #SSPUTV,” later in this subsection).

token-value

is the token value in external representation. Its data representation is determined by the token-type field of *token-code*.

Result

#SSPUT returns a numeric status code indicating the outcome of the SSPUT procedure, as follows:

Code	Condition
0	No error
-1	Invalid buffer format
-2	Invalid parameter value
-3	Missing parameter
-4	Invalid parameter address
-5	Buffer full
-6	Invalid checksum
-7	Internal error
-8	Token not found
-9	Invalid token code or map
-10	Invalid subsystem ID
-11	Operation not supported
-12	Insufficient stack space

Considerations

- You can omit the token-value parameter if the token length specified by *token-code* is zero; otherwise, the token-value parameter is required.
- Specifying a count value greater than 1 is equivalent to calling #SSPUT count times in succession without the COUNT option (but supplying a new token-value before each call).
- If count is greater than 1 and the token is of variable length, each token value must be an even number of bytes in length to ensure word alignment.
- The order in which tokens are added to the buffer is not significant, except in the cases of:
  - #SSPUT calls that start and end lists (using tokens such as ZSPI^TKN^DATALIST, ZSPI^TKN^ERRLIST, and ZSPI^TKN^ENDLIST).
  - A few subsystem-specific exceptions mentioned in the subsystem manuals (the ZEMS^TKN^SUBJECT^MARK token in an event message, for example).

- Adding a token to the buffer with #SSPUT does not affect the current position for subsequent calls to #SSGET or #SSGETV.

Header Tokens and Special Operations for #SSPUT and #SSPUTV

Header tokens for #SSPUT and #SSPUTV include tokens to enable or disable checksum protection, specify the maximum-response value, restore the current position to an earlier value, and specify the server RVU. These tokens are present in the buffer, but have special characteristics that distinguish them from other tokens. Special operations for #SSPUT and #SSPUTV include clearing the last-error information, flushing data from the buffer, deleting tokens, and initializing the current position. These tokens are not in the buffer, but simply serve as parameters to #SSPUT and #SSPUTV.

Table 9-15 lists the header tokens you can supply to #SSPUT(V) to set or change the corresponding values, and also the special tokens you can pass to #SSPUT and #SSPUTV to perform special operations.

Table 9-15. #SSPUT(V) Header Tokens and Special Operations

Token Code	Type	Item retrieved
ZSPI^TKN^CHECKSUM	INT	Enables or disables buffer checksum protection
ZSPI^TKN^CLEARERR	none	Clears last-error information to zero
ZSPI^TKN^DATA^FLUSH	none	Flushes all data in message buffer at or after current position
ZSPI^TKN^DELETE	INT2	Deletes specified token code from buffer
ZSPI^TKN^INITIAL^POSITION	INT	Resets position to start of buffer or start of current list
ZSPI^TKN^MAXRESP	INT	Sets maximum-responses header token
ZSPI^TKN^POSITION	BYTE:8	Restores position saved earlier with #SSGET or #SSGETV
ZSPI^TKN^RESET^BUFFER	UINT	Resets maximum buffer length, clear last-error information, and reset position to start in an SPI buffer received from another process
ZSPI^TKN^SERVER^VERSION	UINT	Sets server-RVU header token
ZSPI^TKN^CHECKSUM	INT	Enables or disables buffer checksum protection

For some of these tokens, when you pass them to #SSPUT or #SSPUTV in the token-code parameter, special considerations apply to other parameters involved.

ZSPI^TKN^CHECKSUM. Use this token code with a nonzero *token-value* to enable checksum protection of the data portion of the buffer; use a zero *token-value* to disable it (the header is always protected by a checksum).

ZSPI^TKN^CLEARERR. Use this token code to clear the last-error information to zero. When using this token code, you must omit *token-value* and the COUNT option. You could use this operation before issuing a series of #SSGET(V) or #SSPUT(V) calls that are followed by a check of the last error. You need this operation only if you use #SSPUT or #SSPUTV to check the header token ZSPI^TKN^LASTERR.

ZSPI^TKN^DATA^FLUSH. Use this token code to flush all information in the message buffer located at or after the current position. You must omit *token-value* and the COUNT option.

ZSPI^TKN^DELETE. Use this token code to delete a token code from the buffer. Specify the token code to be deleted in *token-value*. You can use the SSID option, if needed, to qualify *token-code*. If *token-code* is a token that begins a list, the operation deletes the entire list.

You must use the COUNT option; #SSPUT and #SSPUTV interpret count as the index value of the token code to be deleted.

Note. The ZSPI^TKN^DATA^FLUSH and ZSPI^TKN^DELETE operations do not update the header token ZSPI^TKN^MAX^FIELD^VERSION, which can cause that field to indicate a higher version level than is actually contained in the buffer.

ZSPI^TKN^INITIAL^POSITION. Use this token code to reset the current position as specified by *token-value*. If *token-value* is ZSPI^VAL^INITIAL^BUFFER (zero), the position is reset to the beginning of the buffer. If *token-value* is ZSPI^VAL^INITIAL^LIST (-1), the position is reset to the beginning of the current list.

ZSPI^TKN^MAXRESP. Use this token code to set the header field that specifies the maximum number of responses to return in a single reply message. A *token-value* of zero (the default) specifies one response record per reply, not enclosed in a list. Any *token-value* greater than zero specifies up to that number of response records, each enclosed in a list. A *token-value* of -1 specifies as many response records as can fit, each enclosed in a list.

You must omit the COUNT option.

ZSPI^TKN^POSITION. Use this token code to restore a position previously saved using #SSGET or #SSGETV. The token value is a position descriptor (either eight separate byte values for #SSPUT or an eight-byte STRUCT for #SSPUTV). The buffer contents that precede the previously saved position must not have been modified by ZSPI^TKN^DELETE, ZSPI^TKN^DATAFLUSH, or #SSMOVE operations, or this operation could corrupt the buffer and cause later operations to give unpredictable results. If *token-value* is zero or omitted, this operation sets the current position to the start of the buffer.

ZSPI^TKN^RESET^BUFFER. Use this token before extracting tokens from an SPI buffer received (in either a request or a reply) from another process. This operation performs three actions:

- It resets the maximum buffer length to the value given in *token-value*.

- It clears the last-error information to null values (equivalent to the action of ZSPI^TKN^CLEARERR).
- It resets the current position to the beginning of the buffer (equivalent to the action of ZSPI^TKN^INITIAL^POSITION with ZSPI^VAL^INITIAL^BUFFER).

If *token-value* is less than the actual number of bytes used in the buffer, as given in the header token ZSPI^TKN^USEDLEN, this operation returns an SPI error -5 (buffer full). SPI still resets the maximum buffer length in the SPI message header, causing subsequent calls for that buffer to fail with error -1 (invalid buffer format).

ZSPI^TKN^SERVER^VERSION. Use this token code to set the header field containing the RVU of the server. For *token-value*, supply an unsigned integer that represents the appropriate RVU. For example, if the server is a running RVU C20, specify *token-value* as the unsigned integer 17172, representing a “C” (character number 67) in the left byte (256 x 67 = 17152) and 20 in the right byte.

#SSPUTV Built-In Function

Use #SSPUTV to move binary token values from a variable level into an SPI message buffer. You can use #SSPUTV with any type of token, but you must use #SSPUTV with tokens of type ZSPI^TYP^STRUCT and extensible structured tokens).

```
#SSPUTV [ / option [ , option ]... / ] buffer-var
  token-id source-var
```

option

is either of these:

COUNT *count*

is the token count, an integer in the range 1 through 65535. If greater than 1, count specifies that *source-var* contains an array of count elements, each of which is described by token-id. If you omit this option, it defaults to 1.

If *token-id* is one of the special SPI token codes whose semantics do not allow a token value, you must omit the COUNT option. For the special token code ZSPI^TKN^DELETE, you must supply COUNT; in this case, #SSPUTV interprets count as the index value of the token code to be deleted.

SSID *ssid*

specifies a subsystem ID that qualifies the token code. If you omit this option, or if SSID is zero (0.0.0), it defaults to the subsystem ID of the current list or, if the current position is not in a list, to the subsystem ID specified in the SPI message header (ZSPI^TKN^SSID).

buffer-var

is the name of the SPI message-buffer variable level into which tokens are to be placed.

token-id

is a token code or a token map. It either identifies the token being supplied or denotes a special operation (described in the previous section under “Header Tokens and Special Operations for #SSPUT and #SSPUTV” in the description of the #SSPUT built-in function).

source-var

is the name of the variable level, of type STRUCT, from which #SSPUTV is to obtain the binary token values. The contents of the STRUCT are not altered.

Result

#SSPUTV returns a numeric status code indicating the outcome of the SSPUT procedure, as follows:

Code	Condition
0	No error
-1	Invalid buffer format
-2	Invalid parameter value
-3	Missing parameter
-4	Invalid parameter address
-5	Buffer full
-6	Invalid checksum
-7	Internal error
-8	Token not found
-9	Invalid token code or map
-10	Invalid subsystem ID
-11	Operation not supported
-12	Insufficient stack space

Considerations

- If the token length specified by token-id is zero, you must supply a variable level for *source-var*, but its contents do not matter.
- Specifying a count parameter greater than 1 for #SSPUTV is equivalent to calling #SSPUTV count times without the COUNT option (but supplying a new *token-value* before each call).
- The order in which tokens are added to the buffer is not significant except in the cases of:
  - #SSPUTV calls with token codes for tokens that start and end lists (ZSPI^TKN^DATALIST, ZSPI^TKN^ERRLIST, ZSPI^TKN^LIST, and ZSPI^TKN^ENDLIST).
  - A few subsystem-specific exceptions mentioned in the subsystem manuals (for example, the ZEMS^TKN^SUBJECT^MARK token in an event message).
- If the data in *source-var* is longer than the data area provided by #SSPUTV, the excess bytes are ignored without any notification. If the data in *source-var* is shorter than the data area provided, #SSPUTV sets the remainder of the token value to unspecified values.

- If you specify the COUNT option, #SSPUTV expects the value of *source-var* to be an array of count values of the type of *token-value*. Variable-length token values must be word-aligned.
- Adding a token to the buffer with #SSPUTV does not affect the current position for subsequent calls to #SSGET or #SSGETV.
- When you supply a token map for *token-id*, #SSPUTV uses the version and null-value information in the token map, if necessary, to update the header token ZSPI^TKN^MAX^FIELD^VERSION. The token map is not stored in the buffer; instead, #SSPUTV creates a token code consisting of token type ZSPI^TYP^STRUCT and the token number from the map.
- SPI defines a number of token codes for use with #SSPUT and #SSPUTV to set the values of header tokens and perform special operations. These are described in the Section 8 under “Header Tokens and Special Operations for #SSPUT and #SSPUTV” in the description of the #SSPUT built-in function.

#STOP Built-In Function

Use #STOP to request termination of a process that you started. #STOP passes the name or CPU,PIN of the process to the STOP operating system procedure. #STOP can also pass process deletion message fields that you specify.

```
#STOP [ / option [ , option ] ... / ]
  [ [\node-name.]{$process-name | cpu,pin } [ text ] ]
```

option

can be any of these:

COMPLETIONCODE *num*

specifies the completion code to be placed in the process deletion message; num is a signed integer from -32768 to +32767. Numbers from -32768 to -1 are reserved for use by the TACL software product. Numbers from 0 to 999 are reserved for shared use by both the TACL software product and the user. Numbers from 1000 to 32767 are reserved for use by customers. See the *Guardian Procedure Calls Reference Manual* for a list of defined completion codes.

ERROR

changes the behavior of #STOP as described under “Result.”

SUBSYS *ssid*

identifies the subsystem associated with the process. For information about subsystem ID and other SPI constructs, refer to the *TACL Programming Guide*.

TERMINATIONINFO *num*

specifies information about the termination; num is a signed integer from -32768 to +32767.

node-name

is the name of the system where the process resides.

$*process-name*

is the name of the process.

cpu,pin

indicates the CPU number and process number for the process you want to stop.

text

is text, 1 to 80 characters, for the process deletion message. Leading and trailing spaces are suppressed.

Result

If you do not specify the ERROR option, the #STOP built-in function returns 0 if the STOP system procedure cannot make a successful request to stop the process. If the request is submitted successfully, #STOP returns -1. The designated process might or might not be stopped, depending on the stop mode of the process and the authority of the caller.

If you include the ERROR option, the #STOP built-in function returns the file-system error code passed to it by the STOP system procedure. In this case, zero indicates no error. The #STOP built-in function returns 638 or 639 if the process is queued for stopping but has not actually stopped.

If #STOP stops the TACL process from which you issued it, #STOP returns its result but the TACL process cannot receive it.

Considerations

- A successful #STOP result does not indicate that the process has stopped. A successful result indicates that the stop request was submitted successfully.
- If the process cannot be terminated immediately, the STOP system procedure queues the request.
- If you omit the process designation (process name or CPU,PIN) #STOP stops the current default process (the process last started by TACL or for which TACL most recently paused), if it is still running. If there is no current default process, you must include the process designation. You can examine the default process with the #PROCESS built-in function.
- You must supply the process designation if you wish to include the text argument.
- COMPLETIONCODE, SUBSYS, TERMINATIONINFO, and text are ignored, except when terminating your current TACL process (or, if it is a process pair, either its primary or backup process). If you are stopping your TACL, those items you specify are included in the “stop” system message.

#SUSPENDPROCESS Built-In Function

Use #SUSPENDPROCESS to suspend a process that you started. It invokes the SUSPENDPROCESS operating system procedure.

```
#SUSPENDPROCESS [ [\node-name.]{$process-name | cpu,pin} ]
```

\node-name

is the name of the system where the process resides.

$process-name

is the name of the process.

cpu,pin

indicates the CPU number and process number for the process you want to suspend.

Result

#SUSPENDPROCESS returns a nonzero value if successful; otherwise, it returns zero.

Considerations

- You can omit the process specification (*$process-name* or *cpu,pin*) if a current default process exists. The default process is set by #NEWPROCESS or #PAUSE and clears when that process terminates. You can examine the default process with the #PROCESS built-in function.
- Use #ACTIVATEPROCESS to terminate the suspension.

#SWITCH Built-In Function

Use #SWITCH to make your TACL backup process become the primary process and initialize itself as though you had just logged on to it. The former primary process becomes the backup.

```
#SWITCH
```

Considerations

- Your TACL must have a backup process. To create a backup process, include a backup specification in the #NEWPROCESS call or RUN command, or use the BACKUPCPU command.
- Do not include #SWITCH in an IN file specified in a command to run TACL; if you do so, TACL performs the switch before processing other commands-and each switch causes an initialization-so that the TACL process continues to switch processors.
- #SWITCH establishes an initial logon state, resetting all ASSIGNs, PARAMs, and DEFINEs, invoking the TACLLOCL file and your TACLCSTM file, and setting the history buffer index to 1.

#SYSTEM Built-In Function

Use #SYSTEM to change your current default node name. This function is meaningful only on systems that are in a network.

```
#SYSTEM [ \node-name ]
```

\node-name

specifies the name of the new current default system.

Result

#SYSTEM returns nothing.

Considerations

- If you omit *\node-name*, your saved default system becomes your current default system.
- If you are running a remote TACL process, entering SYSTEM with no following parameters establishes that remote system (not the local system to which your terminal is connected) as your current default system.
- These function calls are not equivalent:

  ```
  14> #SYSTEM \ local-node-name
  ```

  ```
  14> #SYSTEM
  ```

  The first invocation causes the network restrictions on file-name lengths to take effect; the second does not. See the *Expand Network Management and Troubleshooting Guide* for information on network file-name restrictions.

#SYSTEMNAME Built-In Function

Use #SYSTEMNAME to obtain the name of a system, given its network node number.

```
#SYSTEMNAME system-number
```

system-number

is the number of the system whose name is to be found. The system number is an integer in the range 0 to 254.

Result

#SYSTEMNAME returns the name of the system if the system can be reached, -2 if all paths to the system are down, or -1 if the system is not defined.

Consideration

To obtain the network node number of a system, use the #SYSTEMNUMBER built-in function.

Example

These code tests for system availability and displays the node name if the system is available:

```
?SECTION getname MACRO
#PUSH name
#SET name [#SYSTEMNAME %1%]
[#CASE [name]
  | -2 | #OUTPUT System %1% is not available
  | -1 | #OUTPUT System %1% is not defined
  |OTHERWISE| #OUTPUT System %1% (node name [name]) is &
    available
]
```

#SYSTEMNUMBER Built-In Function

Use #SYSTEMNUMBER to obtain the network node number of a system, given its name.

```
#SYSTEMNUMBER \node-name
```

\node-name

is the name of the system whose number is to be found.

Result

#SYSTEMNUMBER returns the number of the system if the system can be reached, -2 if all paths to the system are down, or -1 if the system is not defined.

Consideration

To obtain the name of a system, use the #SYSTEMNAME built-in function.

Example

These code tests for system availability and displays the system number if the system is available:

```
?SECTION getnum MACRO
#PUSH num
#SET num [#SYSTEMNUMBER %1%]
[#CASE [num]
  | -2 | #OUTPUT %1% is not available
  | -1 | #OUTPUT %1% is not defined
  |OTHERWISE| #OUTPUT %1% (system number [num]) is &
    available
]
```

#TACLOPERATION Built-In Function

Use #TACLOPERATION to determine whether TACL is reading commands from IN or $RECEIVE.

```
#TACLOPERATION
```

Result

#TACLOPERATION returns REQUESTER or SERVER depending on whether TACL is receiving its command stream from IN (REQUESTER) or $RECEIVE (SERVER).

#TACLSECURITY Built-In Variable

Use #TACLSECURITY to read the current TACL security, which indicates who can open this TACL process.

```
#TACLSECURITY
```

Result

#TACLSECURITY returns a pair of characters, enclosed in quotes, that represent the current TACL security. The first character represents the criterion that determines whether to allow a process to open the TACL process's $RECEIVE for writing. The second character determines whether to allow an opener with a qualifying name to transfer data to or from a #SERVER. For example:

```
13> #TACLSECURITY
#TACLSECURITY expanded to:
"NN"
```

The characters are the same as for operating system file security. They are as follows:

O allows access only to the Owner on the local system

G allows access to anyone in your Group on your local system

A allows access to Anyone on your local system

U allows access only to the User (owner) on the network

C allows access by any member of your Community on the network

N allows access by anyone on the Network

- allows access only to the local super ID

Considerations

- When you first log on, #TACLSECURITY is initialized to "NN."
- TACL provides an ENQUIRY feature that allows users to obtain the last 22 lines written to a TACL OUT file. To prevent users from accessing data written by a TACL process to its OUT file, set TACL security to an appropriate value.
- Use #PUSH #TACLSECURITY (or PUSH #TACLSECURITY) to save a copy of your current TACL security.
- Use #POP #TACLSECURITY (or POP #TACLSECURITY) to restore the TACL security from the last copy pushed.
- Use #SET #TACLSECURITY (or SET VARIABLE #TACLSECURITY) to define the security applied to write operations to your TACL attempted by other processes.

The syntax of #SET #TACLSECURITY is:

```
#SET #TACLSECURITY "security"
```

"security"

is a two-character security definition as described previously. You must include the quotation marks.

#TACLVERSION Built-In Function

Use #TACLVERSION to determine which RVU of TACL you are using.

```
#TACLVERSION [ / REVISION / ]
```

Result

With the REVISION option, #TACLVERSION returns only the RVU-level portion of the version identifier, followed by a revision level identifier, in the form:

Xnn nnnn

where *Xnn* is the RVU identifier (C20, for example) and *nnnn* is a four-digit number identifying the revision level within that RVU. That number is incremented whenever an interim product modification is made to the RVU.

This form of revision number can be collated. For example, if a macro operation depends on a feature released in a particular RVU of TACL, such as C20 0011, the macro can test to see if the current RVU is equal to or greater than that RVU:

```
[#IF "[#TACLVERSION/REVISION/]" '>=' "C20 0011" |THEN| ...
```

Without the REVISION option, #TACLVERSION returns the current TACL RVU identifier.

Examples

1. This example illustrates the result of #TACLVERSION:

```
17> #OUTPUT [#TACLVERSION]
T9205D30 - 26MAR1999
```

   1. This code disassembles a complete TACL RVU:

```
[#DEF break^version MACRO |BODY|

== First argument is var-level to receive product
== Second argument is var-level to receive version
== Third argument is var-level to receive date

#FRAME

== Make a structure to allow access to individual
== characters:

[#DEF tv STRUCT
BEGIN
 CHAR fld (0:19)
```

```
END;
]

== Put TACL version into the structure

#SET tv [#TACLVERSION]

== Obtain the fields

#SET %1% [tv:fld(0:4)] == T9205 Product number
#SET %2% [tb:fld(5:7)] == Xnn Version

== Ignore 8:10 == " - " (formatting)

#SET %3% [tv:fld(11:19)] == ddMMMyyyy Date

== Clean up and exit

#UNFRAME
]
```

To use this macro, define variables to hold the components of the TACL product number and invoke the macro to assign those components:

```
12> #PUSH product version date
13> BREAK^VERSION product version date
```

#TIMESTAMP Built-In Function

Use #TIMESTAMP to obtain a three-word timestamp from the CPU interval clock in the TACL CPU. The three-word timestamp is also known as a local timestamp and represents the number of centiseconds (.01 second) since 00:00 December 31, 1974.

```
#TIMESTAMP
```

Result

#TIMESTAMP returns the result of the TIMESTAMP operating system procedure.

Consideration

You can convert the result of #TIMESTAMP to a numeric date and time by using #CONTIME.

Example

This example illustrates the #TIMESTAMP result:

```
15> #OUTPUT [#TIMESTAMP]
16> 39950986284
```

#TOSVERSION Built-In Function

Use #TOSVERSION to obtain an identifying letter and number that indicates which RVU of the operating system is running.

```
#TOSVERSION [ \node-name ]
```

\node-name

is the name of an available system containing the operating system whose RVU is to be found.

Result

#TOSVERSION returns the result of the TOSVERSION operating system procedure, in the form character-space-number. For example:

```
27> #TOSVERSION
#TOSVERSION expanded to:
M 20
```

Considerations

- If you omit *\node-name*, #TOSVERSION returns the RVU of the operating system running on the default system.

#TRACE Built-In Variable

Use #TRACE to set or obtain the current state of the trace flag.

```
#TRACE
```

Result

#TRACE returns -1 (true) if the flag is on, 0 (false) if the flag is off.

Considerations

- When you first log on, #TRACE is initialized to zero.
- If the trace flag is on, TACL invokes _DEBUGGER and displays each line before it invokes the line.
- Do not use #TRACE and _DEBUGGER to trace #DELTA commands. Instead, execute #DELTA commands interactively to step through their operation.
- Use #POP #TRACE (or POP #TRACE) to restore the trace flag from the last copy pushed.
- Use #PUSH #TRACE (or PUSH #TRACE) to save a copy of the current trace flag.
- Use #SET #TRACE (or SET VARIABLE #TRACE) to turn the trace flag off or on.
- The syntax of #SET #TRACE is:

```
#SET #TRACE num
```

num

is 0 for off and any nonzero value for on.

#UNFRAME Built-In Function

Use #UNFRAME to pop all variable levels pushed since the last #FRAME.

```
#UNFRAME
```

Result

#UNFRAME returns nothing.

Note. ATTACHSEG operates by pushing and defining a directory variable that refers to the specified segment file; DETACHSEG operates by popping that directory variable. Because #UNFRAME pops all variables pushed since the most recent #FRAME, if you attach a segment file following a #FRAME, the corresponding #UNFRAME detaches the segment file; its contents are no longer available. Subsequent attempts to invoke those contents result in errors.

#USELIST Built-In Variable

Use #USELIST to manage the list of directory variables that you commonly use. If TACL cannot find a specified variable in your home directory, it searches the directories named in the use list.

```
#USELIST
```

Result

#USELIST returns a space-separated list of full path names of the directories in your use list.

Considerations

- When you first log on, #USELIST is initialized to

  ```
  : :UTILS.1 :UTILS.1:TACL.1
  ```

- The use list can contain up to 100 directories.
- The directory variable levels you specify in #SET #USELIST become the new use list; any previous contents are lost. Unlike the USE command, #SET #USELIST does not automatically include directories. Using #SET #USELIST with no parameters removes all directories from the use list; if you do this, you no longer have access to the commands and functions in :UTILS:TACL.
- To keep existing directories and add new ones at the same time, include the existing #USELIST:

  ```
  #SET #USELIST [#USELIST] :newdir
  ```

- Use #PUSH #USELIST (or PUSH #USELIST) to save a copy of the current use list.
- Use #POP #USELIST (or POP #USELIST) to restore the use list from the last copy pushed.
- Use #SET #USELIST (or SET VARIABLE #USELIST) to assign directory names to the use list.

  The syntax of #SET #USELIST is:

  ```
  #SET #USELIST [ directory-name [ directory-name ] ... ]
  ```

  directory-name

  is the name of a directory variable level to be put in the use list.

#USERID Built-In Function

Use #USERID to obtain the user ID (*group-num,user-num*) of a user, given the user name, or to determine whether a specified user is defined on the system.

```
#USERID user
```

user

is either a user name or a user ID.

Result

#USERID returns the user ID of the specified user, if that user is defined; otherwise, it returns nothing.

#USERNAME Built-In Function

Use #USERNAME to obtain the user name of a user, given the user ID (*group-num, user-num*), or to determine whether a specified user is defined on the system.

```
#USERNAME user
```

user

is either a user name or a user ID.

Result

#USERNAME returns the user name of the specified user, if that user is defined on the system; otherwise, it returns nothing.

#VARIABLEINFO Built-In Function

Use #VARIABLEINFO to obtain information about a variable level.

```
#VARIABLEINFO / option [ , option ] ... / variable-level
```

option

specifies the type of information requested; it can be any of these:

DEPTH

returns the total number of levels for the specified variable.

DIRECTORY

returns the full name of the directory that contains the specified variable.

EXISTENCE

returns -1 if the variable level exists, 0 otherwise.

FRAME

returns the frame number of the specified variable.

LEN

returns the length of the specified variable level, as modified by its type.

LEVEL

returns the absolute number of the specified variable level. If you do not specify a level number, the absolute number of the top level is given.

LINES

returns a line count for the specified variable level, as modified by its type.

MODE

returns the I/O mode, if any, of the specified variable level. The I/O modes are IN, DYNAMIC_IN, OUT, STATUS, PROMPT, READ, WRITE, and ERROR.

If the variable level is a directory whose variables are in a different segment file from that in which the directory resides, MODE returns the access mode to that segment file, PRIVATE or SHARED.

OCCURS

returns the number of occurrences of characters, substructures, or simple items within the specified variable level, depending on its type.

OFFSET

returns the number of bytes between the beginning of the structure and the first byte of data of the specified variable level (if the specified variable level is a STRUCT item).

PROCESS

returns the process identification associated with the specified variable level by means of an implicit server.

- For an unnamed process, a process ID (that is, *cpu,pin)* is returned.
- For a named process, the process name is returned.
- If the process name is not available, nothing is returned.

REQUESTER

returns the file name associated with the specified variable level, if it is used as a requester variable.

SEGMENT

returns the name of the segment file that contains the variables of a directory, if *variable-level* is a directory and its variables are in a different segment file from that in which *variable-level* resides. In all other cases, this option is inoperative.

SERVER

returns the server process name associated with the specified variable level, if that variable level is used as an explicit server variable.

TYPE

returns the type of the variable level.

VARIABLE

returns the name of the specified variable, stripped of any level-number identification.

variable-level

is the name of the variable level about which you are requesting information.

Result

#VARIABLEINFO returns a space-separated list of the requested information about the variable level.

In the case of the LEN, LINES, OCCURS, OFFSET, and TYPE options, the information returned is a function of the type of variable level involved.

Table 9-16 lists the result for each of these options depending on the type of argument given to #VARIABLEINFO.

Table 9-16. #VARIABLEINFO Type-Dependent Results

Variable Type	LEN	LINES	OCCURS	OFFSET	TYPE
Alias	0	0	0	0	ALIAS
Delta	1	(a)	(b)	0	DELTA
Directory	0	0	0	0	DIRECTORY
Macro	1	(a)	(b)	0	MACRO
Routine	1	(a)	(b)	0	ROUTINE
Structure	(c)	1	1	0	STRUCT
Structure Item	(d)	1	(e)	(f)	(g)
Text	1	(a)	(b)	0	TEXT

(a) Number of text lines in the variable level.

(b) Number of characters in the variable level, with each line end counting as a character.

(c) Number of data bytes required to hold the entire structure.

(d) Number of data bytes required to hold one occurrence of the substructure or simple item.

(e) Number of occurrences of the substructure or simple item.

(f) Number of bytes before the first byte of data of the first occurrence of the specified substructure or simple item, regardless of its declared bounds.

(g) STRUCT for a substructure, or the data type for a simple item. (See Section 3, “Variables,” for a list of data types.)

Considerations

- The information is returned in the order in which the options are given.
- If the specified variable level does not exist, all options except EXISTENCE return nothing.
- When you apply the LINES or OCCURS option to a DELTA, MACRO, ROUTINE, or TEXT variable level, the entire variable level must be read to compute the specified values.
- OCCURS and LEN ignore bounds specifications when operating with a STRUCT item.
- Each logical line within a variable contains an internal end-of-line character that counts as one byte. For variables that contain TACL statements, each square bracket ([,]), vertical bar (|), or tilde-space combination (~_) uses two bytes, including unprintable characters that are subject to change from one TACL RVU to another. Other characters use one byte.
- If you are using #SETMANY to assign a number of results from #VARIABLEINFO to several variable levels, put those options that can return empty results at the end of the options to prevent losing the correlation between the two lists.

#VARIABLES Built-In Function

Use #VARIABLES to obtain the names of all variables in your home directory or the fully qualified names of certain selected variable levels.

```
#VARIABLES [ / { BREAKPOINT | IO } / ]
```

BREAKPOINT

specifies that only those variable levels for which debugging breakpoints have been set are to be listed.

IO

specifies that only the “I/O variables” (variable levels used by processes, requesters, and servers) are to be listed.

Result

- If you do not specify an option, #VARIABLES returns a space-separated list of the unqualified names of all variables in your home directory.
- If you include the BREAKPOINT option, #VARIABLES returns a space-separated list of the fully qualified names of all variable levels that have breakpoints set on them.
- If you include the I/O option, #VARIABLES returns a space-separated list of the fully qualified names of the specific variable levels used by processes, #SERVERs, and #REQUESTERs.

Considerations

- The variable name list is not alphabetically ordered, but you can use the #SORT built-in function to make it so.
- To obtain the names of variables and store them into a variable level, use the #VARIABLESV built-in function.

#VARIABLESV Built-In Function

Use #VARIABLESV to obtain the names of some or all of the variables in your home directory and to put them into a variable level, each name on a separate line. The selection criteria are the same as for #VARIABLES.

```
#VARIABLESV [ / { BREAKPOINT | IO } / ] variable-level
```

BREAKPOINT

specifies that only those variable levels for which debugging breakpoints have been set are to be listed.

IO

specifies that only the “I/O variables” (variable levels used by processes, requesters, and servers) are to be listed.

variable-level

is an existing variable level that is to receive the names of the variables. The names are placed in the variable level one per line, in an unspecified order.

Result

#VARIABLESV returns nothing.

Considerations

- Placing a list of variables in *variable-level* causes its previous contents, if any, to be lost.
- This function is most useful when a very large number of variables causes the #VARIABLES function to produce a “text buffer overflow” error.
- To obtain the names of variables as the result of a function call, use the #VARIABLES built-in function.

#WAIT Built-In Function

Use #WAIT to specify the variable level(s) for which you require TACL to wait. TACL does not execute the next instruction until the variable is ready. The conditions that define ready depend on the purpose for which the variable level is used and are listed under “Considerations.”

```
#WAIT variable-level [ variable-level ] ...
```

variable-level

is the name of a variable level for which the routine must wait.

Result

#WAIT returns the name and level number of the first variable level that is ready.

Considerations

These guidelines apply to the use of the #WAIT built-in function:

- These criteria determine whether a variable level is ready:
  - If a variable level is not in use by a process, #SERVER, or #REQUESTER, the #WAIT built-in function always considers it ready.
  - If a variable level is in use by a #SERVER or the DYNAMIC IN of a process, it is ready if it is empty (all data has been sent and the process is waiting for input).
  - If a variable level is in use by a #SERVER or the OUT or STATUS of a process, it is ready if it is not empty (data or status is available).
  - If a variable level is in use as the error variable of a #REQUESTER, it is ready if it is not empty (an error has occurred). TACL sets such a variable level only if an error occurs, not if the I/O operation is successful.
  - If a variable level is in use by a READ #REQUESTER read variable, it is ready if it is not empty (data is available).
  - If a variable level is in use by a WRITE #REQUESTER write variable, it is ready if it is empty (all data has been written).
- If you want to ensure that #WAIT always returns immediately, create an additional variable level, not used for I/O, and specify its name last in the WAIT list.
- To access only the topmost level of a variable, regardless of its level number, you could invoke the #WAIT function as follows:

```
#PUSH error^var read^var prompt^var
[#CASE [#VARIABLEINFO /VARIABLE/
       [#WAIT error^var read^var prompt^var] ]
```

```
  |error^var| == Handle error^var ready condition here
  |read^var| == Handle read^var ready condition here
  |prompt^var| == Handle prompt^var ready condition here
]
```

- When waiting for a prompt variable, be sure to clear the variable before initiating the read and #WAIT operations. Otherwise, the variable might be ready despite the outcome of your operation. For an example, see the *TACL Programming Guide*.
- To provide thorough error handling when using processes and files accessed by #REQUESTER or #SERVER, wait for all related variables.

#WAKEUP Built-In Variable

Use #WAKEUP to set or obtain the current state of the WAKEUP flag. If the WAKEUP flag is on, #PAUSE (or PAUSE) ends when any process you started stops, instead of when the last process you started stops.

```
#WAKEUP
```

Result

#WAKEUP returns the current state of the WAKEUP flag: -1 if the flag is on, 0 if it is off.

Considerations

- When you first log on, #WAKEUP is initialized to zero.
- Use #PUSH #WAKEUP (or PUSH #WAKEUP) to save a copy of the current setting of the WAKEUP flag.
- Use #POP #WAKEUP (or POP #WAKEUP) to restore the WAKEUP flag setting from the last copy pushed.
- Use #SET #WAKEUP (or SET VARIABLE #WAKEUP) to set the state of the WAKEUP flag.

  The syntax of #SET #WAKEUP is:

  ```
  #SET #WAKEUP num
  ```

num

is -1 to set the flag on and 0 to set it off.

#WIDTH Built-In Variable

Use #WIDTH to set or obtain the current setting of the width register, which indicates the maximum width of lines TACL writes to its OUT file.

```
#WIDTH
```

Result

#WIDTH returns the current setting of the width register.

Considerations

- When you first log on, #WIDTH is initialized to 80.
- Use #PUSH #WIDTH (or PUSH #WIDTH) to save a copy of the current width setting.
- Use #POP #WIDTH (or POP #WIDTH) to restore the width register from the last copy pushed.
- Use #SET #WIDTH (or SET VARIABLE #WIDTH) to define the maximum number of characters that TACL is to output before starting a new line.

  The syntax of #SET #WIDTH is:

  ```
  #SET #WIDTH num
  ```

  num

  is a number in the range 10 to 239.

- For D-series software RVUs, only the default number of characters or number of characters set using the #set #width command are displayed. Excess characters are truncated. For the default width (set to 80):

  ```
  $DATA08 USERX 15> time

  February 21, 2001 16:32:49
  ```

  For G-series software RVUs, the number of characters indicates the number of characters to be displayed on a line. Excess characters are displayed on a subsequent line, and so forth. For width set to 10:

  ```
  $DATA08 USERX 16> #set #width 10
  $DATA08 USERX 17> time

  February 2
  1, 2001 16
  :32:58
  ```

#XFILEINFO Built-In Function

#XFILEINFO is the built-in function that implements the FILEINFO command. Its syntax and use are exactly the same as those of the FILEINFO command.

#XFILENAMES Built-In Function

#XFILENAMES is the built-in function that implements the FILENAMES command. Its syntax and use are exactly the same as those of the FILENAMES command.

#XFILES Built-In Function

#XFILES is the built-in function that implements the FILES command. Its syntax and use are exactly the same as those of the FILES command.

#XLOADEDFILES Built-In Function

#XLOADEDFILES is the built-in function that implements the LOADEDFILES command. Its syntax and use are exactly the same as those of the LOADEDFILES command.

#XLOGON Built-In Function

#XLOGON is the built-in function that implements the LOGON command. Its syntax and use are exactly the same as those of the LOGON command.

#XPPD Built-In Function

#XPPD is the built-in function that implements the PPD command. Its syntax and use are exactly the same as those of the PPD command.

#XSTATUS Built-In Function

#XSTATUS is the built-in function that implements the STATUS command. Its syntax and use are exactly the same as those of the STATUS command.

For example:

```
#XSTATUS *, LOADED <FILENAME>
```

will return all the processes which are using the given LOAD file.

A Syntax Summary

The syntax diagrams summarized in this appendix are divided into five categories:

- The command interpreter set of commands and functions, supplied with TACL in the directory :UTILS:TACL
- The built-in functions and variables that constitute the TACL programming language
- The specialized forms of the #DEF function used to create and redefine structured variables (STRUCT declarations)
- The specialized forms of the #SET function used to assign values to TACL built-in variables
- The commands of the #DELTA character processor

Differences between H-series and G-series command syntax are noted in this section.

Differences between D-series and G-series, if any, may be found in the detailed syntax contained in Section 8, UTILS:TACL Commands and Functions.

:UTILS:TACL Commands and Functions

The following summarizes the syntax of the TACL command interpreter commands and functions:

```
ACTIVATE [ [\node-name.]{$process-name | cpu,pin } ]

ADD DEFINE
  { define-name |( define-name [, define-name ]...) }
  [, LIKE define-name ] [, attribute-spec ] ...}

ADDDSTTRANSITION start-date-time , stop-date-time , offset

ADDUSER [ / run-option [ , run-option ] ... / ]
  group-name.user-name , group-id,user-id

ALARMOFF

ALTER DEFINE define-name-list
  { { , attribute-spec } | { , RESET reset-list } }

ALTPRI [\node-name.]{$process-name | cpu,pin } , pri

ASSIGN [ logical-unit [ , [ actual-file-name ]
  [ , create-open-spec ] ... ] ]

ATTACHSEG { PRIVATE | { SHARED file-name directory-name } }

BACKUPCPU [ cpu ]

BREAK [ variable-level ]

BUILTINS [ / { FUNCTIONS | VARIABLES } / ]

BUSCMD [ / run-option [ , run-option ] ... / ]
{ X | Y } , { DOWN | UP } , from-cpu , to-cpu

CLEAR
  ALL  | ALL ASSIGN | ALL PARAM | ASSIGN logical-unit |
  PARAM param-name

COLUMNIZE list

COMMENT [ comment-text ]

_COMPAREV string-1 string-2

COMPUTE expression

_CONTIME_TO_TEXT contime-list

_CONTIME_TO_TEXT_DATE contime-list

_CONTIME_TO_TEXT_TIME contime-list

COPYDUMP [ / run-option [ , run-option ] ... / ]
  source-file , dest-file
```

COPYDUMP is not supported on H-series systems.

```
COPYVAR variable-level-in variable-level-out
```

```
CREATE file-name [ , extent-size ]

CREATESEG file-name

DEBUG [ [\node-name.]{$process-name | cpu,pin } ]
  [ , TERM [\node-name.]$terminal-name ]

_DEBUGGER current-trace-value reason-for-entry

DEFAULT [ / run-option [ , run-option ] ... / ] default-names
  [,"default-security"] ,"default-security"

DELETE DEFINE define-name-list

DELUSER [ / run-option [ , run-option ] ... / ]
  group-name.user-name

DETACHSEG directory-name

ENV [ environment-parameter ]

EXIT

FC [ num | -num | text ]

FILEINFO [ / OUT list-file / ]
  [ file-name-template [ [,] file-name-template ] ... ]

FILENAMES [ / OUT list-file / ]
  [ file-name-template [ [,] file-name-template ] ... ]

FILES [ / OUT list-file / ]
  [ subvol-template [ [,] subvol-template ] ... ]

FILETOVAR file-name variable-level

HELP

HISTORY [ num ]

HOME [ directory-name ]

INFO [ / OUT list-file / ] DEFINE define-name-list
  [, DETAIL ]

INITTERM

INLECHO { OFF | ON }

INLEOF

INLOUT { OFF | ON }

INLPREFIX [ prefix ]

INLTO [ variable-level ]

JOIN variable-level

KEEP [ / LIST / ] num variable [ variable ] ...

KEYS
```

```
LIGHTS [ / run-option [, run-option ] .../ ]

  [ ON  | OFF | SMOOTH [ num ] ]
  [ , sys-option | , ALL ]
  ... [, BEAT ]

LOAD [ / KEEP num / ] file-name [ file-name ] ...

LOGOFF [ / option [, option ] ... / ]

LOGON
{ group-name.user-name | group-id,user-id | alias }

[ [ ,password ] |
[ , old-password, new-password ] |

[ , old-password, new-password, new-password ]]
[ ; parameter [, parameter ] ...]

_LONGEST list

_MONTH3 num

O[BEY] command-file

OUTVAR [ / option [ , option ] ... / ] string

PARAM [ param-name param-value
  [ , param-name param-value ] ... ]

PASSWORD [ / run-option [ , run-option ] ... / ]
  [ password ]

PAUSE [ [\node-name.]{$process-name | cpu,pin } ]

PMSEARCH subvol-spec [ [,] subvol-spec ] ...

PMSG { ON | OFF }

POP variable [ [,] variable ] ...
```

H-Series POSTDUMP

```
POSTDUMP <in file> [ < options > ]

PPD [ / OUT list-file / ]
  [ [\node-name.][ { $process-name | cpu,pin | * } ] ]

PURGE / option / file-name-template [, file-name-template...]

PUSH variable [ [,] variable ] ...
```

G-Series RCVDUMP

```
RCVDUMP [ / run-option [ , run-option ] ... / ]
  dump-file , cpu , { X | Y } [ , param [ , param ] ]
```

H-Series RCVDUMP

```
RCVDUMP   <filename>, cpuNum [, SLICE <sliceId>]
[, START <startAddress>][, END <endAddress>]
[ [, ONLINE | PARALLEL] [, FULL] ]
```

RECEIVEDUMP is not supported on H-series systems.

G-Series RECEIVEDUMP

```
RECEIVEDUMP / OUT dump-file / cpu , fabric
  [ , param [ , param ] ]

RELOAD [ / run-option [, run-option ] ... / ]
  [ [ cpu-set [; cpu-set ]... ]

REMOTEPASSWORD [ \node-name [ , password ] ]

RENAME old-file-name [,] new-file-name

RESET DEFINE
  { { attribute-name [, attribute-name ]... } | * }

[ RUN[D] ] program-file [ / run-option [ , run-option ].../ ]
  [ param-set ]

SEGINFO

SET DEFINE
  { attribute-spec | LIKE define-name }[, attribute-spec ]...

SET DEFMODE { ON | OFF }

SET HIGHPIN { ON | OFF }

SET INSPECT { OFF | ON | SAVEABEND }

SETPROMPT { SUBVOL | VOLUME | BOTH | NONE }

SET SWAP [ $volume-name ]

SETTIME
  { { month day } | { day month } } year , hour: min[: sec]
  [  GMT | LST |LCT ]

SET VARIABLE [ / option [, option ]/ ] variable-level
  [ text ]

SET VARIABLE built-in-variable [ built-in-text ]

SHOW [ / OUT list-file / ] [ attribute [ , attribute ] ... ]

SHOW [ / OUT list-file / ] DEFINE [ attribute-name | * ]

SINK [ text ]

STATUS [ / OUT list-file / ] [ range ] [ , condition ] ...
  [ , DETAIL ] [ , STOP ]

STOP [ [\node-name.]{$process-name | cpu,pin } ]

SUSPEND [ [\node-name.]{$process-name | cpu,pin } ]

SWITCH

SYSTEM [ \node-name ]

SYSTIMES

[\node-name.]TACL [ / run-option [ , run-option ] ... / ]
[ backup-cpu-num ] [ ; parameter [ , parameter ] ]
```

```
TIME

USE [ directory-name [ [,] directory-name ] ... ]

USERS [ / run-option [ , run-option ] ... / ] [ range ]

VARIABLES [ directory-name ]

VARINFO [ variable [ [,] variable ] ... ]

VARTOFILE variable-level file-name

VCHANGE [ / option [ , option ] ... / ] variable-level
  string-1 string-2 [ range ]

VCOPY [ / option [ , option ] ... / ] source-var range
  dest-var dest-line

VDELETE [ / option [ , option ] / ... ] variable-level range

VFIND [ / option [ , option ] / ... ] variable-level string
  [ range ]

VINSERT variable-level line-num

VLIST [ / option [ , option ] / ...] variable-level [ range ]

VMOVE [ / option [ , option ] / ... ] source-var range
  dest-var dest-line

VOLUME [ [\node-name.] volume ] [ , "security" ]

VTREE [ directory-name ]

WAKEUP { ON | OFF }

WHO

{ X | Y }BUSDOWN from-cpu , to-cpu

{ X | Y }BUSUP from-cpu , to-cpu

! [ num | - num | text ]

? [ num | - num | text ]
```

Built-In Functions and Variables

The following summarizes the syntax of the built-in functions and variables used for programming in TACL:

```
#ABEND [ / option [ , option ] ... / ]
  [ [\node-name.]{$process-name | cpu,pin } [ text ] ]

#ABORTTRANSACTION

#ACTIVATEPROCESS [ [\node-name.]{$process-name | cpu,pin} ]

#ADDDSTTRANSITION low-gmt high-gmt offset

#ALTERPRIORITY [ [\node-name.]{$process-name | cpu,pin } ]
  pri

#APPEND to-variable-level [ text ]

#APPENDV to-variable-level { from-variable-level | string }

#ARGUMENT [ / option [ , option ] ... / ]

alternative [ alternative ] ...

#ASSIGN [ / option [ , option ] ... / logical-unit ]

#BACKUPCPU [ cpu ]

#BEGINTRANSACTION

#BREAKMODE

#BREAKPOINT variable-level state

#BUILTINS [ / { FUNCTIONS | VARIABLES } / ]

#CASE text enclosure

#CHANGEUSER [ / CHANGEDEFAULTS / ]
  { group-name.user-name | group-id,user-id | alias }
  password

#CHARACTERRULES

#CHARADDR variable-level line-addr

#CHARBREAK variable-level char-addr

#CHARCOUNT variable-level

#CHARDEL variable-level char-addr-1
  [ FOR char-count | TO char-addr-2 ]

#CHARFIND [ / EXACT / ] variable-level char-addr text

#CHARFINDR [ / EXACT / ] variable-level char-addr text

#CHARFINDRV [ / EXACT / ] variable-level char-addr string

#CHARFINDV [ / EXACT / ] string-1 char-addr string-2
```

```
#CHARGET variable-level char-addr-1
  [ FOR char-count | TO char-addr-2 ]

#CHARGETV var-1 var-2 char-addr-1
  [ FOR char-count | TO char-addr-2 ]

#CHARINS string char-addr text

#CHARINSV variable-level char-addr string

#COLDLOADTACL

#COMPAREV string-1 string-2

#COMPUTE expression

#COMPUTEJULIANDAYNO year month day

#COMPUTETIMESTAMP year month day hour min sec milli micro

#COMPUTETRANSID system cpu sequence crash-count

#CONTIME timestamp

#CONVERTPHANDLE
  {  / PROCESSID / integer-string } |
  { / INTEGERS / process-identifier }

#CONVERTPROCESSTIME process-time

#CONVERTTIMESTAMP gmt-timestamp direction [ \node-name ]

#CREATEFILE [ / option [ , option ] / ] file-name

#CREATEPROCESSNAME

#CREATEREMOTENAME \node-name

#DEBUGPROCESS [ / NOW / ]
  [\node-name.]{$process-name | cpu,pin }
  [ , TERM [\node-name.]$terminal-name ]

#DEF variable
{ { ALIAS | DELTA | MACRO | ROUTINE |TEXT } enclosure } |
  { {DIRECTORY [ segment-spec ]} | {STRUCT structure-body} }

#DEFAULTS [ / option [ , option ] / ]

#DEFINEADD define-name [ flag ]

#DEFINEDELETE define-name

#DEFINEDELETEALL

#DEFINEINFO define-name

#DEFINEMODE

#DEFINENAMES define-template

#DEFINENEXTNAME [ define-name ]

#DEFINEREADATTR
  { define-name | _ } { attribute-name | cursor-mode }
```

```
#DEFINERESTORE [ / option [ , option ] ... / ] buffer

#DEFINERESTOREWORK

#DEFINESAVE [ / WORK / ] define-name buffer

#DEFINESAVEWORK

#DEFINESETATTR attribute-name [ attribute-value ]

#DEFINESETLIKE define-name

#DEFINEVALIDATEWORK

#DELAY csecs

#DELTA [ / COMMANDS variable-level / ] [ text ]

#DEVICEINFO / option [ , option ] ... /
  { $device-name | file-name }

#EMPTY [ text ]

#EMPTYV [ / BLANK / ] string

#EMSADDSUBJECT [ / SSID ssid / ] buffer-var
  token-id [ token-value ]

#EMSADDSUBJECTV [ / SSID ssid / ] buffer-var
  token-id source-var

#EMSGET [ / option [ , option ] ... / ] buffer-var get-op

#EMSGETV [ / option [ , option ] ... / ] buffer-var get-op
  result-var

#EMSINIT [ / option [ , option ] / ] buffer-var ssid
  event-number token-id [ token-value ] ... ]

#EMSINITV [ / option [ , option ] / ] buffer-var ssid
  event-number token-id source-var

#EMSTEXT [ / option [ , option ] ... / ] buffer-var

#EMSTEXTV [ / option [ , option ] ... / ] buffer-var
  formatted-var [ lengths-var ]

#ENDTRANSACTION

#EOF variable-level

#ERRORNUMBERS

#ERRORTEXT / option [ option ] ... /

#EXCEPTION

#EXIT

#EXTRACT variable-level

#EXTRACTV from-variable-level to-variable-level
```

```
#FILEGETLOCKINFO [ / option / ] fvname control lockdesc
  participants

#FILEINFO / option [ , option ] ... / file-name

#FILENAMES [ / option [ , option ] ... / ]
  [ file-name-template ]

#FILTER [ exception [ exception ] ... ]

#FRAME

#GETCONFIGURATION / option [ , option ] ... /

#GETPROCESSSTATE [ / option [ , option ] ... / ]

#GETSCAN

#HELPKEY

#HIGHPIN

#HISTORY [ / option [ , option ] ... / ]

#HOME

#IF [ NOT ] numeric-expression [ enclosure ]

#IN

#INFORMAT

#INITTERM

#INLINEECHO

#INLINEEOF

#INLINEOUT

#INLINEPREFIX

#INLINEPROCESS

#INLINETO

#INPUT [ / option [ , option ] ... / ] [ prompt ]

#INPUTEOF

#INPUTV [ / option [ , option ] ... / ] variable-level
  prompt-string

#INSPECT

#INTERACTIVE [ / CURRENT / ]

#INTERPRETJULIANDAYNO julian-day-num

#INTERPRETTIMESTAMP four-word-timestamp

#INTERPRETTRANSID transid

#JULIANTIMESTAMP [ type [ tuid-request ] ]
```

```
#KEEP num variable

#KEYS

#LINEADDR variable-level char-addr

#LINEBREAK variable-level line-addr char-offset

#LINECOUNT variable-level

#LINEDEL variable-level line-addr-1
  [ FOR line-count | TO line-addr-2 ]

#LINEFIND [ / EXACT / ] variable-level line-addr text

#LINEFINDR [ / EXACT / ] variable-level line-addr text

#LINEFINDRV [ / EXACT / ] variable-level line-addr string

#LINEFINDV [ / EXACT / ] variable-level line-addr string

#LINEGET string line-addr-1
  [ FOR line-count | TO line-addr-2 ]

#LINEGETV string variable-level line-addr-1
  [ FOR line-count | TO line-addr-2 ]

#LINEINS variable-level line-addr text

#LINEINSV variable-level line-addr string

#LINEJOIN variable-level line-addr

#LOAD [ / option [ , option ] / ] file-name

#LOCKINFO lock-spec tag buffer

#LOGOFF [ / option [ , option ] ... / ]

#LOOKUPPROCESS / option [ , option ] ... / specifier

#LOOP enclosure

#MATCH template [ text ]

#MOM

#MORE

#MYGMOM

#MYPID

#MYSYSTEM

#MYTERM

#NEWPROCESS program-file [ / option [, option ]... / ]
  [ param-set ]

#NEXTFILENAME [ file-name ]

#OPENINFO / option [ , option ] / { file-name | device-name }
  tag
```

```
#OUT

#OUTFORMAT

#OUTPUT [ / option [ , option ] ... / ] [ text ]

#OUTPUTV [ / option [ , option ] ... / ] string

#PARAM [ param-name ]

#PAUSE [ [\node-name.]{$process-name | cpu,pin } ]

#PMSEARCHLIST

#PMSG

#POP variable [ [,] variable ] ...

#PREFIX

#PROCESS

#PROCESSEXISTS [\node-name.]{$process-name | cpu,pin }

#PROCESSFILESECURITY

#PROCESSINFO / option [ , option ] ... /
  [ [\node-name.]{$process-name | cpu,pin } ]

#PROCESSORSTATUS [ \node-name ]

#PROCESSORTYPE [ / BOTH | NAME / ]
  { { [\node-name.]{$process-name | cpu, pin } } | cpu-num

#PROMPT

#PURGE file-name

#PUSH variable [ [,] variable ] …

#RAISE exception

#REPLYV string

#REQUESTER [ / option [ , option ] / ]
{ CLOSE variable-level }

{ READ file-name error-var read-var prompt-var } |
{ WRITE file-name error-var write-var }

#RESET option [ option ] …

#REST

#RESULT [ text ]

#RETURN

#ROUTEPMSG { ALL | STANDARD |
( message-type [ message-type ] ... ) }

#ROUTINENAME

#SEGMENT [ / USED / ]
```

```
#SEGMENTCONVERT / FORMAT { a | b } / old-file-name
  new-file-name

#SEGMENTINFO / option [ , option ] / [ segment-id ]

#SEGMENTVERSION file-name

#SERVER / option [ , option ] ... / [ server-name ]

#SET
  { [ / option [, option ]/] variable-level [ text ] } |
  { built-in-variable [ built-in-text ] }

#SETBYTES destination source

#SETCONFIGURATION / option [, option ] .../[ tacl-image-name]

#SETMANY variable-name-list , [ text ]

#SETPROCESSSTATE
  / LOGGEDON | TSNLOGON | STOPONLOGOFF | PROPAGATELOGON |
    PROPAGATESTOPONLOGOFF / { 0 | 1 }

#SETSCAN num

#SETSYSTEMCLOCK julian-gmt mode [ tuid ]

#SETV dest-variable-level source-string

 #SHIFTDEFAULT

#SHIFTSTRING [ / option / ] [ text ]

#SORT [ / option / ] [ text ]

#SPIFORMATCLOSE

#SSGET [ / option [ , option ] ... / ] buffer-var get-op

#SSGETV [ / option [ , option ] ... / ] buffer-var get-op
  result-var

#SSINIT [ / TYPE 0 / ] buffer-var ssid command
  [ / type-0-option [ , type-0-option ] ... / ] token-id

#SSNULL token-map struct

#SSPUT [ / option [ , option ] ... / ] buffer-var token-id
  [ token-value [ token-value ] ... ]

#SSPUTV [ / option [ , option ], / ] buffer-var token-id
  source-var

#STOP [ / option [ , option ] ... / ]

[ [\node-name.]{$process-name | cpu,pin } [ text ] ]

#SUSPENDPROCESS [ [\node-name.]{$process-name | cpu,pin} ]

#SWITCH

#SYSTEM [ \node-name ]

#SYSTEMNAME system-number
```

```
#SYSTEMNUMBER \node-name
#TACLOPERATION
#TACLSECURITY
#TACLVERSION / REVISION /
#TIMESTAMP
#TOSVERSION [ \node-name ]
#TRACE
#UNFRAME
#USELIST
#USERID user
#USERNAME user
#VARIABLEINFO / option [ , option ] ... / variable-level
#VARIABLES [ / { BREAKPOINT | IO } / ]
#VARIABLESV [ / { BREAKPOINT | IO } / ] variable-level
#WAIT variable-level [ variable-level ] ...
#WAKEUP
#WIDTH
```

STRUCT Declarations

The following summarizes the forms of the #DEF function used to create and redefine structured variables:

```
#DEF variable STRUCT
  { BEGIN declaration [ declaration ] ... END ; }
  |
  ( LIKE structure-identifier ; }

type identifier [ VALUE initial-value ] ;

type identifier ( lower-bound : upper-bound )
  [ VALUE initial-value ] ;

STRUCT identifier [ ( lower-bound : upper-bound ) ] ;
  { BEGIN declaration [ declaration ] ... END ; }
  |
  { LIKE structure-identifier ; }

FILLER num ;

type identifier [ ( lower-bound : upper-bound ) ]
  REDEFINES previous-identifier ;
```

#SET Summary

The following summarizes the syntax of the #SET function when it is used to assign values to built-in variables. SET VARIABLE commands used for the same purpose have the same syntax.

```
#SET #ASSIGN [ [ / option [ , option ] ... / ] logical-unit ]
#SET #BREAKMODE { DISABLE | ENABLE | POSTPONE }
#SET #CHARACTERRULES file-name
#SET #DEFAULTS subvolume-name
#SET #DEFINEMODE { OFF | ON }
#SET #ERRORNUMBERS n n n n
#SET #EXIT num
#SET #HELPKEY [ key-name ]
#SET #HIGHPIN { OFF | ON }
#SET #HOME directory
#SET #IN file-name
#SET #INFORMAT { PLAIN | QUOTED | TACL }
#SET #INLINEECHO num
#SET #INLINEOUT num
#SET #INLINEPREFIX [ prefix ]
#SET #INLINETO [ variable-level ]
#SET #INPUTEOF num
#SET #INSPECT { OFF | ON | SAVEABEND }
#SET #MYTERM home-term
#SET #OUT file-name
#SET #OUTFORMAT { PLAIN | PRETTY | TACL }
#SET #PARAM [ param-name [ param-value ] ]
#SET #PMSEARCHLIST searchlist
#SET #PMSG num
#SET #PREFIX [ text ]
#SET #PROCESSFILESECURITY " security"
#SET #PROMPT num
#SET #REPLYPREFIX [ num]
#SET #SHIFTDEFAULT { DOWN | NOOP | UP }
```

```
#SET #TACLSECURITY "security"
#SET #TRACE num
#SET #USELIST [ directory-name [ directory-name ] ... ]
#SET #WAKEUP num
#SET #WIDTH num
```

#DELTA Command Summary

Table A-1 summarizes the syntax of the #DELTA character processor commands.

Table A-1. #DELTA Commands (page 1 of 3)

Command	Description
*x*A	Convert ASCII
y,*x*A	Convert ASCII with error return
B	Beginning
*x*C	Character move
x:C	Character move with return code
*x*D	Delete
EI*file*$	Open file for input
EO*file*$	Open file for output
*x*FC	Lowercase lines
y,*x*FC	Lowercase characters
x@FC	Uppercase lines
y,*x*@FC	Uppercase characters
FE*var*$	Test variable level for emptiness
FF*var*$	Get frame number of variable level
*x*FG*var*$	Compare lines to variable level
y,*x*FG*var*$	Compare range to variable level
FL	Get length from last I or S operation
FO*var*$	Pop variable
FT*var*$	Get variable type
*x*FT*var*$	Set variable type
FU*var*$	Push variable
*x*FU*var*$	Push and load variable with x
G*var*$	Get text from variable level
H	Whole buffer
I*text*$	Insert text

Table A-1. #DELTA Commands (page 2 of 3)

Command	Description
*x*I	Insert ASCII
y,*x*I	Insert y*ASCII
*x*J	Jump characters
*x*K	Kill lines
y,*x*K	Kill characters
*x*L	Move by lines
M*var*$	Invoke macro
*x*P	Write lines
y,*x*P	Write characters
Q*var*$	Get value from variable level
*x*S*text*$	Search
x:S*text*$	Search with return code
*x*T	Type lines
y,*x*T	Type characters
@T*var*$	Type variable level contents
:T*text*$	Type text
*x*U*var*$	Unload x into variable level
y,*x*U*var*$	Unload x into variable level
*x*V	View lines
x:V	View lines and show end
*x*X*var*$	Extract lines to variable level
y,*x*X*var*$	Extract characters to variable level
*x*Y	Read lines
Z	Get buffer size
\	Convert number in text to value in X
x\	Put x in text
^\	Exit from macro
?	Condition
:?	NOT condition
'	End condition
,	Move X into Y
$	Clear X and Y
.	Get current position
=	Display X or Y,X

Table A-1. #DELTA Commands (page 3 of 3)

Command	Description
<	Begin iteration
x<	Iterate x times
;	Exit iteration
>	End iteration
@>	End iteration, do not decrement iteration count
!	Comment

B Error Messages

TACL can generate several types of errors. Table 5-1 on page 5-21 lists the types of TACL errors and a sample display, description, and typical action for each type of error.

The first subsection describes:

- TACL Error Messages - the text and meaning of each TACL error message and warning.

Following subsections describe more specialized error messages :

- DEFINE Error Messages on page B-46
- Process Creation Error Messages on page B-50
- RCVDUMP Error Messages on page B-51
- RELOAD Error Messages on page B-58
- EMS Messages on page B-69

Finally, Table B-2 on page B-70 lists error numbers that can be obtained from function calls, including a call to #ERRORNUMBERS. The explanations of those errors can be found in the alphabetic listing.

△ **Caution.** TACL text messages might change at anytime. You should not write TACL macros or routines that are dependent on the format of text messages.

TACL Error Messages

General TACL error messages are preceded by *ERROR*. Some of the messages in the following list may appear following another error message to give additional information; when that occurs, the messages appear in parentheses, without the *ERROR* prefix.

```
ABENDED: $XX
CPU time: 0:00:00.018
Termination Info: 24
TACL fatal error: super-group privilege required
```

Cause. For a RUN command with the PORTTACL option, the user must have a 255 group ID.

Effect. The TACL process ABENDs (terminates abnormally).

Recovery. Use a LOGON with a user group ID of 255 and execute the RUN command again.

```
*ERROR* All block buffers in use
```

Cause. Certain built-in functions that perform I/O, including (#)PUSH #IN, (#)PUSH #OUT, and #REQUESTER, require block buffers. There are only four available. More than four block buffers would have been needed to carry out the action you requested.

Effect. The requested operation is ignored.

Recovery. Perform a (#)POP #IN or #OUT or #REQUESTER /CLOSE/ operation as necessary.

```
*ERROR* All possible simultaneous servers and requesters in
use
```

Cause. You attempted to create more than 100 simultaneous requesters and servers (explicit and implicit).

Effect. The requested operation is ignored.

Recovery. You will have to use no more than 100 simultaneous requesters and servers.

```
*ERROR* A non-STRUCT may appear only at the end
```

Cause. You attempted to use a simple item at a place other than the end of a structure reference.

Effect. The requested operation is ignored.

Recovery. Correct the structure reference.

```
*ERROR* Arithmetic overflow
```

Cause. An operation has caused a processor-detected arithmetic overflow. (Be sure you did not try to divide by zero.)

Effect. The requested operation is ignored.

Recovery. There is a pointer that helps you to isolate the place where the overflow occurred. Make the needed correction and retry the operation.

```
*ERROR* Backup CPU is down
```

Cause. In a (#)BACKUPCPU request, you specified that the TACL backup is to be run in a processor that is down.

Effect. The requested operation is ignored.

Recovery. Specify a different CPU, or wait until the desired processor is back in operation.

```
*ERROR* Backup CPU may not be same as primary CPU
```

Cause. In a (#)BACKUPCPU request, you specified that the TACL backup is to be run in the same CPU as its primary.

Effect. The requested operation is ignored.

Recovery. Specify a different processor as the backup CPU.

```
*ERROR* Backup process already exists
```

Cause. You attempted to start a backup process for your TACL, but a backup process exists already.

Effect. No process is created.

Recovery. A process cannot have more than one backup. If your intent is to move your TACL backup to another CPU, you must first delete the existing backup process by issuing a BACKUPCPU command with no argument.

```
*ERROR* Backup process device subtype differs from primary
```

Cause. The system tried to create a process, but the backup process device subtype is not the same as the primary process device subtype.

Effect. No process is created.

Recovery. Correct the call to #NEWPROCESS. This error can occur if the process you started tried to create its backup using a program file that has a process subtype different from that of the program file being run by the primary. In this case, the error can be corrected only by debugging the program.

```
*ERROR* Badly formed label
```

Cause. You specified a label in the wrong form.

Effect. The requested operation is ignored.

Recovery. Check the syntax for the operation you are trying to perform and correct the label format accordingly.

```
*ERROR* Badly formed variable name
```

Cause. You formatted a variable name incorrectly.

Effect. The requested operation is ignored.

Recovery. Change the variable name to the correct format.

```
*ERROR* BODY label not found
```

Cause. #DEF did not contain the required |BODY| label.

Effect. The requested operation is ignored.

Recovery. Add the required |BODY| label to the #DEF function. #DEF requires |BODY| if the type specified is MACRO, ROUTINE, TEXT, ALIAS, or DELTA. |BODY| is not allowed if the type is STRUCT or DIRECTORY.

```
*ERROR* Cannot create Process File Segment
```

Cause. Insufficient lockable memory for the process file segment.

Effect. The process was not started.

Recovery. Reissue the #NEWPROCESS command with a different CPU number, or stop other processes that are using lockable memory in the CPU.

```
*ERROR* Cannot push or pop the root segment's root
```

Cause. You attempted to perform a #DEF, (#)KEEP, (#)LOAD, (#)POP, or (#)PUSH operation on the root (:) directory.

Effect. The requested operation is ignored. If you specified the #DEF or (#)PUSH functions, TACL might lose user variables and TACL variables.

Recovery. Remove all attempts to use these commands and functions with the root segment directory. Specify the entity to be defined, kept, loaded, pushed, or popped as something other than the root.

```
*ERROR* Cannot resolve alias
```

Cause. There was an error in an alias variable.

Effect. The requested operation is ignored.

Recovery. Check that you have correctly indicated the command for which the variable is an alias.

```
*ERROR* Cannot retrieve I/O process info from a C-series node
```

Cause. You attempted a PPD process, * operation for an I/O process on a C-series node.

Effect. TACL displays the error.

Recovery. Remove all attempts to retrieve I/O process information from C-series nodes.

```
*ERROR* Conflicting floating point type in object files
```

Cause. You used a floating-point format in your program that conflicts with the floating-point format in your user library object file.

Effect. The operating system does not allow the creation of the process.

Recovery. Resolve the conflict.

```
*ERROR* CONVERTPROCESSTIME argument too large (>3.7 years)
```

Cause. A call to the CONVERTPROCESSTIME operating system procedure included a request to convert more than 3.7 years.

Effect. The requested operation is ignored.

Recovery. Correct the call so that the maximum process execution time does not exceed 3.7 years.

```
*ERROR* CPRULES file is corrupt
```

Cause. Indeterminate.

Effect. The requested operation is ignored.

Recovery. See your service provider to obtain a new copy of the CPRULES file in question.

```
*ERROR* CPRULES file must be file code 199
```

Cause. You specified a CPRULES file with a file code other than 199. (TACL also accepts a file code of 180 for the CPRULES file.)

Effect. The requested operation is ignored.

Recovery. Determine the correct file name.

```
*ERROR* DEFINE context error
```

Cause. An error occurred when the NEWPROCESS or PROCESS_CREATE_ procedure attempted to propagate existing DEFINEs to the new process.

Effect. No process is created.

Recovery. Use #ERRORNUMBERS to see the subcode (*nnn*) that indicates the specific cause of the error:

Subcode	Meaning
0	Unable to convert a DEFINE name to network form.
1	Indicates that flags.<10> = 1 and flags.<11:12> <> 0 and you are trying to create a process on a remote system that is a NonStop 1+, or any other NonStop system operating under operating system RVU B20 or earlier; the use of DEFINEs is not allowed.
2	Internal context propagation error.
3	Illegal DEFMODE supplied.

Adjust and retry the call to #NEWPROCESS. If errors recur, contact your service provider.

```
*ERROR* Delta syntax error
```

Cause. The #DELTA command you entered had a syntax error.

Effect. The command is ignored.

Recovery. Correct the command and reissue the request.

```
*ERROR* Device does not exist [ , File: file-name ]
```

Cause. You selected a device that is not on the system. *file-name* indicates the file specified.

Effect. The requested operation is ignored.

Recovery. Check that you have the correct device name. Check your defaults.

```
*ERROR* Directory contents cannot be set
```

Cause. You attempted to perform a (#)SET operation on a variable whose type is DIRECTORY.

Effect. The requested operation is ignored.

Recovery. Specify a variable of a type other than DIRECTORY in the (#)SET operation.

```
*ERROR* Duplicate exception in this statement
```

Cause. You specified the same exception more than once in a #FILTER.

Effect. The requested operation is ignored.

Recovery. Delete the duplicate exception in your #FILTER.

```
*ERROR* Duplicate keyword
```

Cause. You specified the same keyword twice in a place where duplicate keywords are not allowed.

Effect. The requested operation is ignored.

Recovery. Consult the syntax description for the feature you were trying to use. Delete the unneeded keyword and reissue the request.

```
*ERROR* Enclosure not allowed in #DEF of DIR or STRUCT
```

Cause. You specified an enclosure in a #DEF function call that defines a directory variable or a STRUCT variable.

Effect. TACL displays the error.

Recovery. Remove the enclosure from the definition.

```
*ERROR* Enclosure not allowed in this context
```

Cause. You specified an enclosure in a statement other than a #CASE, #DEF, #LOOP, or #IF function call.

Effect. TACL displays the error.

Recovery. Remove the enclosure from the statement.

```
*ERROR* Enclosure not allowed in unbracketed line
```

Cause. You specified an unbracketed line that contains an enclosure. This error can occur if you use a vertical line as a text character and #INFORMAT is set to TACL, causing TACL to interpret it as a special character.

Effect. The requested operation is ignored.

Recovery. Add brackets to the line containing your enclosure. Precede the vertical line with a tilde (~|) to specify that it is to be treated as ordinary text.

```
*ERROR* Expecting a legal subvolume name
```

Cause. Misspelling of #DEFAULTS in a #PMSEARCHLIST expression.

Effect. The operation fails.

Recovery. Correct the spelling to “#DEFAULTS”.

```
*ERROR* Expecting a number or an arithmetic expression
```

Cause. For a RUN command, the PFS <num-pages> value is out of range.

Effect. The RUN command fails.

Recovery. Specify a correct value, from 64 through 512.

```
*ERROR* Expecting |DO|
```

Cause. You called #LOOP with |WHILE| but did not include |DO|.

Effect. The requested operation is ignored.

Recovery. Rewrite the loop to include |DO| after |WHILE|.

```
*ERROR* Expecting |THEN| or |ELSE|
```

Cause. You called #IF but did not include |THEN| or |ELSE|.

Effect. The requested operation is ignored.

Recovery. Rewrite the statement to include |THEN| or |ELSE| after #IF.

```
*ERROR* Expecting |UNTIL|
```

Cause. You called #LOOP with |DO| but did not include |UNTIL|.

Effect. The requested operation is ignored.

Recovery. Rewrite the loop to include |UNTIL| after |DO|.

```
*ERROR* Expecting |WHILE| or |DO|
```

Cause. You called #LOOP without including |DO| or |WHILE|.

Effect. The requested operation is ignored.

Recovery. Rewrite the loop to include |DO| or |WHILE| after #LOOP.

```
*ERROR* Extended data segment initialization error
```

Cause. A file-system error occurred while you were setting up the extended data segment of a program you are trying to run.

Effect. The extended data segment is not initialized.

Recovery. Check that you have specified a valid volume

```
*ERROR* Extended data segment swap file error
```

Cause. A file-system error occurred while you were setting up the swap file of a program you are trying to run. A secondary message following this one gives the file name.

Effect. The swap file is not created.

Recovery. Check that you have specified your swap file correctly.

```
*ERROR* File error on previous out file
```

Cause. A file-system error occurred on a pushed OUT file.

Effect. TACL pops #OUT.

Recovery. Check the file-system error that led to this message and handle that error accordingly. The *Guardian Procedure Errors and Messages Manual* contains descriptions of file-system error messages.

```
*ERROR* File: file-name
```

Cause. This is a secondary message that names the file involved in an error described by the message that immediately precedes this one.

Effect. See the description of the primary message.

Recovery. See the description of the primary message.

```
*ERROR* File has undefined data blocks
```

Cause. You attempted to use an object file that contains C data variables that were declared with the EXTERN declaration but that were not defined.

Effect. The requested operation is ignored.

Recovery. Define the variable and include a data block with the same name in the object file.

```
*ERROR* File in use [ , File: file-name ]
```

Cause. The specified file is already open, and shared access is not permitted. If you get this error with a process name error when creating a new process, another process is running under that name.

Effect. The file is not opened for you.

Recovery. Find out why the file is open, then close it if possible, or wait until the other user is finished with the file. You might be trying to open the file twice. If the message refers to a process, use a different process name.

```
*ERROR* File is full [ , File: file-name ]
```

Cause. No more records could be added to the file.

Effect. The requested operation is ignored.

Create a new file with larger extents and reload the file. See the description of creating files with FUP in the *File Utility Program (FUP) Reference Manual*.

```
*ERROR* File is neither a program nor a TACL macro
```

Cause. You attempted to invoke a file that is neither a program nor a macro file.

Effect. The requested operation is ignored.

Recovery. Check that you have named the correct file.

```
*ERROR* File not fixed-up by binder
```

Cause. You attempted to build an object file using BINSERV or BIND in conjunction with the ?FIXUPS OFF directive or the SELECT FIXUPS OFF command.

Effect. The requested operation is ignored.

Recovery. If BINDER does not fix code and data references in the object file, it cannot be run. Bind the file again without turning off fixing. Fixups are applied by default.

```
*ERROR* File not in directory or record not in file [ , File:
file-name ]
```

Cause. You specified a file name that could not be found or attempted to read a record that was not in the file.

Effect. The requested operation is ignored.

Recovery. Check that you have the right file.

```
*ERROR* File number has not been opened [ , File: file-name ]
```

Cause. A file-system call used a file number that was not assigned to an open file.

Effect. The requested operation is ignored.

Recovery. Report the problem to your service provider.

```
*ERROR* File or record already exists [ , File: file-name ]
```

Cause. When creating or renaming a file, the name specified was already in use on the specified subvolume.

Effect. The requested operation is ignored.

Recovery. Select a new name for the new file, or purge or rename the existing file.

```
*ERROR* File system error occurred on library file
```

Cause. The system monitor encountered a file-system error while accessing the library file during process creation.

Effect. The requested operation is ignored.

Recovery. Check that you have specified the library file correctly and then retry the request. The file you specified is not a TACL library file; it is an object file referred to by a program file.

```
*ERROR* GUARDIAN call to TEXTTOSSID failure xxx
```

Cause. Errors encountered by the #SSINIT built-in function can result from the TACL product calling the Guardian procedure TEXTTOSSID.

Effect. The function fails.

Recovery. Recovery is determined by the context of the error message detail. Refer to the description of the TEXTTOSSID procedure in the *Guardian Procedure Calls Reference Manual* for a description of the possible errors.

```
*ERROR* GUARDIAN File Error: nnn
```

Cause. A file-system error occurred for which TACL has no descriptive message.

Effect. The requested operation is ignored.

Recovery. See the *Guardian Procedure Errors and Messages Manual* for corrective action for the error number indicated by *nnn*.

```
*ERROR* GUARDIAN SETMYTERM failed
```

Cause. TACL could not change its home terminal when you tried to set #MYTERM.

Effect. The requested operation is ignored.

Recovery. Use a terminal name acceptable to the operating system.

```
*ERROR* Header INITSEGS not consistent with size
```

Cause. The file was not produced by BINDER, or BINDER malfunctioned.

Effect. The requested operation is ignored.

Recovery. Contact your service provider.

```
*ERROR* Hometerm error
```

Cause. When starting a process, you specified an illegal home terminal, or the specified home terminal does not exist.

Effect. The requested operation is ignored.

Recovery. Check that you have correctly identified the home terminal.

```
*ERROR* IEEE floating point support not available on CPU
```

Cause. You used IEEE floating-point format in program and attempted to run it in a processor that does not support IEEE floating-point format.

Effect. The operating system does not allow the creation of the process.

Recovery. The PROCESSOR_GETINFOLIST_ Guardian procedure call can be used to determine if a processor can run IEEE floating-point instructions. For more information about this procedure, see the *Guardian Procedure Calls Reference Manual*.

```
*ERROR* Illegal CPU number
```

Cause. You specified a CPU number that is not in the range 0 through 15.

Effect. The requested operation is ignored.

Recovery. Specify a CPU number within the allowed range.

```
*ERROR* Illegal filename specification [ , File: file-name ]
```

Cause. A file name was specified that was not in the correct file system form.

Effect. The requested operation is ignored.

Recovery. See the appropriate command or function description in this manual for information about the correct *file-name* format.

```
*ERROR* Illegal internal character representation
```

Cause. Internally, TACL represents certain special characters as two bytes, the first having the value 255. This error results if TACL detects an unexpected value in the second byte. This value is likely to occur only if you use #DELTA without due caution when editing a variable that contains TACL statements.

Effect. The requested operation is ignored.

Recovery. Use care when editing a variable containing TACL statements.

```
*ERROR* Illegal level number syntax
```

Cause. You specified a level number in an incorrect form. A level number consists of a period and a number, which can be either a literal number or a numeric variable. If you use a variable name, it must not include any directories, nor can it include a level number.

Effect. The requested operation is ignored.

Recovery. Correct or remove the level-number specification.

```
*ERROR* Illegal library file format
```

Cause. When you tried to start a new process, the library file associated with the program file failed the tests that ensure that the file is actually a library. (The file is not a TACL library.)

Effect. The requested operation is ignored.

Recovery. Check that you specified the correct file in the LIB option when you started the process. If you did not specify a library file, someone else may have bound in the incorrect file name using LIB, or the file may have been altered.

```
*ERROR* Illegal process device subtype
```

Cause. #NEWPROCESS attempted to run an object file with an illegal device subtype. The device subtype is an attribute stored in each object file. The process created from an object file is assigned the device subtype stored in that object file. Only named processes can have nonzero device subtypes.

Device subtypes in the range 1 through 15 are reserved for processes that are:

- Created by the super ID
- Created from licensed object files
- Created from object files owned and provided by the super ID

Effect. No process is created.

Recovery. Verify that the call to #NEWPROCESS contains the name parameter. If the name parameter exists, change the device subtype to an unrestricted value, or contact the super ID.

```
*ERROR* Illegal program file format
```

Cause. An attempt was made to run a program, but the program file failed one of the tests that ensure that the file is actually a program.

Effect. The requested operation is ignored.

Recovery. Check that you specified the correct file.

```
*ERROR* Illegal value
```

Cause. You attempted to assign an invalid value to a built-in variable (for example, you cannot give the value #SET to the built-in variable #INLINEPREFIX).

Effect. The requested operation is ignored.

Recovery. See the description of the built-in variable in question, in Section 9, Built-In Functions and Variables.

```
*ERROR* Illegal variable name syntax
```

Cause. You specified a variable name that does not have the correct form.

Effect. The requested operation is ignored.

Recovery. Check the syntax of the variable name and correct it.

```
*ERROR* In OBEY file file-name: error-information
```

Cause. TACL detected an error while processing an OBEY file.

Effect. Depends on the error; TACL attempts to continue processing the file.

Recovery. Review the error information that accompanies this error and correct the problem. If the error is a process creation error, see Process Creation Error Messages on page B-50.

```
*ERROR* Invalid alias format
```

Cause. A variable of type ALIAS can contain only a single variable name or a single file name.

Effect. The requested operation is ignored.

Recovery. Correct the alias variable.

```
*ERROR* Invalid comment format
```

Cause. You entered a comment in an incorrect format.

Effect. The requested operation is ignored.

Recovery. The proper comment formats are the COMMENT command or the comment characters (braces or double equal signs).

```
*ERROR* Invalid PEP
```

Cause. The file was not produced by BINDER, or BINDER malfunctioned, resulting in an invalid procedure entry point.

Effect. The requested operation is ignored.

Recovery. Contact your service provider.

```
*ERROR* Invalid username or password
```

Cause. When trying to log on, you gave an incorrect user name or password.

Effect. The requested operation is ignored.

Recovery. Reissue the LOGON command with the correct user name and password.

```
*ERROR* I/O error writing to terminal
```

Cause. An I/O error occurred at the home terminal. Undefined externals exist in the object file that #NEWPROCESS is attempting to run, so the procedure cannot open, or write to, the home terminal to display the undefined-externals message.

Effect. No process is created.

Recovery. The file-system error code can be found by the use of #ERRORNUMBERS. See the *Guardian Procedure Errors and Messages Manual* for corrective action for that error number.

```
*ERROR* Item name already in use
```

Cause. You attempted to reuse an item name already used at the same level of the current structure.

Effect. The requested operation is ignored.

Recovery. Use an item name that is not already in use.

```
*ERROR* Item not found in STRUCT
```

Cause. You attempted to perform a REDEFINE operation on an item that is not defined at the same level of the current structure.

Effect. The requested operation is ignored.

Recovery. Check that you have spelled the name correctly. If not, correct it; if so, ensure that the item is defined, at the same level of the current structure, before you refer to it in a redefinition.

```
*ERROR* Level is in use
```

Cause. You attempted to associate a variable level with more than one server file or #REQUESTER.

Effect. The requested operation is ignored.

Recovery. A particular variable level can be associated with no more than one server file or #REQUESTER at a time.

```
*ERROR* LIB file has main procedure
```

Cause. You specified a user library object file with a MAIN procedure in it.

Effect. The requested operation is ignored.

Recovery. Use the LIST command in BIND to determine the MAIN procedure present. The LIST command also specifies whether the MAIN attribute is set. Either remove the MAIN procedure or set the MAIN attribute to OFF by using the ALTER command in BINDER.

```
*ERROR* Library conflict
```

Cause. When trying to run a program, you specified a library file that was not the same as the library file used by another copy of the program. All processes running a given program must use the same library. This is an operating system problem.

Effect. The requested operation is ignored.

Recovery. Check that you specified the correct library file.

```
*ERROR* LIKE may not refer to any part of STRUCT containing
it
```

Cause. You attempted to define a structure or item LIKE a structure containing the LIKE clause.

Effect. The requested operation is ignored.

Recovery. Ensure that the LIKE clause refers to another structure, not the one in which the LIKE appears.

```
*ERROR* LOGON has been disabled. Log off first.
```

Cause. You attempted to log on from a logged-on terminal and the NOCHANGEUSER field of the TACL configuration does not equal zero.

Effect. The requested operation is ignored.

Recovery. Log off before attempting to log on.

```
*ERROR* Maximum number of attached segment files exceeded
segment-file-name not attached
```

Cause. You specified more than the maximum number of allocated segment files.

Effect. The requested operation is ignored.

Recovery. Reduce the number of segment files and reissue the request.

```
*ERROR* Missing close bracket
```

Cause. You omitted a closing bracket.

Effect. The requested operation is ignored.

Recovery. Supply the missing bracket and reissue the request.

```
*ERROR* Missing close quote; must be on same line as open
quote
```

Cause. You either forgot the closing quotation mark of a string or attempted to include an end-of-line delimiter in a string.

Effect. The requested operation is ignored.

Recovery. Supply the closing quotation mark and reissue the request.

```
*ERROR* Missing open bracket
```

Cause. You omitted an opening bracket.

Effect. The requested operation is ignored.

Recovery. Supply the missing bracket and reissue the request.

```
*ERROR* Multiply defined label in enclosure
```

Cause. The same label was present more than once in an enclosure.

Effect. The requested operation is ignored.

Recovery. Either delete or modify the extra copies of the label.

```
*ERROR* Name of variable, builtin, file, or system file
needed
```

Cause. TACL cannot find anything to invoke by the name you entered.

Effect. The requested operation is ignored.

Recovery. Check that you spelled the name correctly or that you used the name of an existing variable, built-in, file, or system file. Then reissue the request.

```
*ERROR* Neither case label nor OTHERWISE found
```

Cause. The case label could not be found, and there was no OTHERWISE included in the CASE statement.

Effect. The requested operation is ignored.

Recovery. Check that you specified the case label correctly, or add an OTHERWISE.

```
*ERROR* NEWPROCESS Error: nnn, nnn
```

Cause. When you attempted to create a new process, an error occurred for which TACL has no explanatory text.

Effect. The requested operation is ignored.

Recovery. See the *Guardian Procedure Errors and Messages Manual* for corrective action for the error indicated by the first number; see the section on file system errors in that manual for corrective action for the error indicated by the second number.

```
*ERROR* No backup
```

Cause. You attempted to do a (#)SWITCH operation while your TACL had no backup process.

Effect. The requested operation is ignored. Ensure that your TACL has a backup before requesting a (#)SWITCH operation.

Recovery. Ensure that your TACL has a backup before requesting a (#)SWITCH operation.

```
*ERROR* No data pages
```

Cause. You attempted to execute a program with less than one data page. Either there was no global or local data, or data pages were set using the compiler directive ?DATAPAGES or the BIND command SET DATAPAGES.

Effect. The requested operation is ignored.

Recovery. Use the SHOW command with the SET option in BIND to determine the number of data pages in the object file. Use the CHANGE command in BIND to set the number of pages.

```
*ERROR* No level number or STRUCT qualification allowed
```

Cause. You supplied a level number, or a reference to a STRUCT item or substructure, where TACL accepts only an unqualified variable name.

Effect. The requested operation is ignored.

Recovery. Check the variable reference to ensure that it refers to the correct entity, in the proper form.

```
*ERROR* No main procedure
```

Cause. You attempted to execute a program that either is missing a main procedure or that has the MAIN attribute set to OFF.

Effect. The requested operation is ignored.

Recovery. Use the LIST command with the CODE option in BIND to verify that the MAIN procedure was not present in the object file, or that the MAIN attribute for the procedure was set to OFF. Either add the MAIN procedure or set the MAIN attribute to ON by using the ALTER command in BINDER.

```
*ERROR* Nondirectory may only be at end of variable name
```

Cause. You attempted to follow a name that is not of type DIRECTORY with a colon and another variable name.

Effect. The requested operation is ignored.

Recovery. Ensure that the entity to which you are referring is defined as type DIRECTORY, or remove the directory reference.

```
*ERROR* Nonexistent directory in variable name
```

Cause. You attempted to follow a nonexistent variable name with a colon and another variable name.

Effect. The requested operation is ignored.

Recovery. Define the desired directory before referring to it.

```
*ERROR* No PCB available
```

Cause. The system monitor could not create a process control block (PCB) because all entries in the PCB table are in use.

Effect. The requested operation is ignored.

Recovery. Try running the program on a different CPU, or wait until a PCB entry becomes free.

```
*ERROR* No routine has been called
```

Cause. You attempted to invoke one of certain built-in functions, such as #RESULT, that can be called only from within a routine, but the function call is not in a routine.

Effect. The requested operation is ignored.

Recovery. Include the invocation of the built-in function in a routine.

```
*ERROR* No such line
```

Cause. The line requested with the HISTORY command does not exist. It could be a line not yet used (number too large) or a line no longer in the history buffer (number too small), or the text string does not match a line in the buffer.

Effect. The requested operation is ignored.

Recovery. Check that you specified the correct line. Use the HISTORY command to check for the line you want

```
*ERROR* No such variable
```

Cause. You tried to invoke a variable that does not exist.

Effect. The requested operation is ignored.

Recovery. Change the variable to one that exists.

```
*ERROR* Not a disk file
```

Cause. The file specified is not a disk file.

Effect. The requested operation is ignored.

Recovery. Check that you specified the correct file. If not, retry the operation with the correct file; otherwise, corrective action depends on the application.

```
*ERROR* Not a legal variable type or IO mode
```

Cause. You attempted to use a variable or I/O mode that is not allowed.

Effect. The requested operation is ignored.

Recovery. Check that you have correctly specified the variable type or I/O mode.

```
*ERROR* Not correct file structure
```

Cause. The file specified has an incorrect structure.

Effect. The requested operation is ignored.

Recovery. Check that you specified the correct file. If not, retry the operation with the correct file; otherwise, corrective action depends on the application.

```
*ERROR* Not file code 100
```

Cause. The file specified has a code other than 100.

Effect. The requested operation is ignored.

Recovery. Check that you specified the correct file. If not, retry the operation with the correct file; otherwise, corrective action depends on the application.

```
*ERROR* Not found
```

Cause. The requested search string could not be found. This message results from the #DELTA S command.

Effect. The requested operation is ignored.

Recovery. Correct the search string if you know it does exist.

```
*ERROR* Not in buffer
```

Cause. This condition can result from the #DELTA C, L, T, K, D, and J commands. The indicated characters are not in the buffer.

Effect. The requested operation is ignored.

Recovery. Use the #DELTA HT command to display the contents of the buffer.

```
*ERROR* Number or numeric variable expected
```

Cause. The variable indicated must be or contain a number.

Effect. The requested operation is ignored.

Recovery. Correct the indicated variable.

```
*ERROR* One or more errors occurred while loading library
file
```

Cause. There was a problem with a TACL library file that you tried to load.

Effect. The requested operation is ignored.

Recovery. Check your library file and correct the errors.

```
*ERROR* Option conflicts with another option
```

Cause. You specified an option that conflicts with another option; for example, specifying IN and INV in the same RUN command or #NEWPROCESS call.

Effect. The requested operation is ignored.

Recovery. Correct the conflict and reissue the request.

```
*ERROR* Option may not appear more than once
```

Cause. You specified more than one option or specified the same option more than once with the indicated request.

Effect. The requested operation is ignored.

Recovery. Delete the additional option and reissue the request.

```
*ERROR* |OTHERWISE|, if present, must be the last label in
#CASE
```

Cause. An OTHERWISE label is not last in a #CASE built-in function.

Effect. The function stops executing.

Recovery. Move the OTHERWISE label to the end of the #CASE function.

```
*ERROR* Process File Segment size out of range
```

Cause. You specified an invalid number of pages for the PFS option of a #NEWPROCESS call.

Effect. The process was not started.

Recovery. Reissue the RUN or #NEWPROCESS command with PFS num-pages in the allowable range.

```
*ERROR* PROCESS_GETINFO_ error = <n>, error^detail = <d>
```

Cause. TACL received error <*n*> unexpectedly from operating system procedure PROCESS_GETINFO_.

Effect. For each of these errors, #ERRORNUMBERS is set:

First number	1150, 1151, 1152, or 1153
Second Number	<*n*>
Third number	<*d*>
Fourth number	0

Recovery. Refer to the *Guardian Procedure Errors and Messages Manual* for a description of PROCESS_GETINFO_ errors. The error information and the context in which the error occurs may suggest a recovery action. If not, contact your service provider.

```
*ERROR* #PROCESSLAUNCH built-in not supported in this TOS
version
```

Cause. The operating system RVU predates the D40 RVU and does not support the system procedure PROCESS_LAUNCH_:

Effect. The specified process is not started.

Recovery. None. Either upgrade the system to use the function or do not use it.

```
*ERROR* Process name error
```

Cause. When attempting to start or modify a process, you specified a process name that is invalid because the name exists already or the name was incorrectly specified.

Effect. The requested operation is ignored.

Recovery. Check that you specified the process name correctly. Use the STATUS command to see if the process name exists already and who owns the process.

```
*ERROR* Program file and library file specified are same
file: text
```

Cause. When trying to run a program, you specified the same name for the program and the library. This problem is detected by the operating system.

Effect. The requested operation is ignored.

Recovery. Try the command again with the correct program and library names.

```
*ERROR* Program file error: text
```

Cause. When you attempted to run a program, the program file could not be used for the reason indicated in text.

Effect. The requested operation is ignored.

Recovery. Check that you specified the correct program file.

```
*ERROR* Range must be last
```

Cause. You can specify a range only as the last item in a structure reference.

Effect. The requested operation is ignored.

Recovery. Reformat the structure reference to place the range after the name of the item containing the range.

```
*ERROR* RECORD size cannot exceed 1024 bytes for unstructured
non-edit files
```

Cause. The #REQUESTER built-in function opened an unstructured non-edit file with a WAIT option of greater than 1024 bytes specified for the record length.

Effect. The requested operation is ignored.

Recovery. Make the original record size smaller.

```
*ERROR* REDEFINE item would extend beyond item being
redefined
```

Cause. You supplied a redefinition whose data area would be larger than the data area of the item being redefined, starting at offset 0 in the item being redefined.

Effect. The requested operation is ignored.

Recovery. Make the original definition larger, to accommodate the redefinition, or make the redefinition smaller.

```
*ERROR* REDEFINES attempted to place word-aligned item on
odd-byte boundary
```

Cause. A REDEFINES clause in a STRUCT declaration called for the placement of a word-aligned item on an odd-byte boundary.

Effect. The requested operation is ignored.

Recovery. Use a FILLER byte to ensure proper alignment of a STRUCT item and another item that redefines it.

```
*ERROR* Reference out of defined bounds
```

Cause. You attempted to refer to a structure element outside its defined range. You can get this error if you define the range of an element so that it does not contain 0,and you then refer to that element without giving a specific offset.

Effect. The requested operation is ignored.

Recovery. Check the defined range of the structure element and either correct the range reference to fall within the definition or adjust the definition to accommodate the reference.

```
*ERROR* Rename attempted to another volume [ , File: file-
name ]
```

Cause. You attempted to rename a file to another disk volume, which is not possible.

Effect. The requested operation is ignored.

Recovery. Use FUP DUP to copy the file to the other disk volume, then purge the original.

```
*ERROR* Requires a numeric argument
```

Cause. The #DELTA command you entered requires a numeric expression.

Effect. The requested operation is ignored.

Recovery. Correct the command and reissue the request.

```
*ERROR* Requires a specific level
```

Cause. You failed to append a level number to a variable name when TACL was expecting a level specification.

Effect. The requested operation is ignored.

Recovery. Supply the needed level specification.

```
*ERROR* Requires later version of GUARDIAN
```

Cause. Most often caused by attempting to execute a program using BINDER features that are not supported by the RVU of the operating system in use.

Effect. The requested operation is ignored.

Recovery. Check that the RVU of BINDER in use is of the same or earlier RVU as the operating system in use. If the RVU of BINDER in use is from a later RVU than that of the operating system, it may include features that are not supported.Rebind the object file, using a RVU of BINDER that is of the same RVU as the operating system in use. For additional information, see the *Binder Manual*.

```
*ERROR* Resident size greater than code area
```

Cause. The file was not produced by BINDER, or BINDER malfunctioned.

Effect. The requested operation is ignored.

Recovery. Contact your service provider.

```
*ERROR* Routine stack overflow
```

Cause. You attempted to use calls between routines resulting in more than 100 layers.

Effect. The requested operation is ignored.

Recovery. Reduce the number of recursive routines.

```
*ERROR* Security violation [ , File: file-name ]
```

Cause. You attempted an operation on a file that has a protection specifier that does not allow the action requested.

Effect. The requested operation is ignored.

Recovery. Use the FILEINFO command to check who owns the file and how the protection specifier is set. If you own the file, change the protection specifier on the file (if necessary).

```
*ERROR* Segment data structure invalid
```

Cause. You attempted to attach a segment file whose file code is 440 but whose contents are not a valid TACL segment.

Effect. The requested operation is ignored.

Recovery. Check whether you are referring to the correct file; if so, check the file contents. Repair as the situation indicates.

```
*ERROR* Segment file code must be 440
```

Cause. You attempted to attach a segment file whose file code is not 440, the only valid file code for TACL segment files.

Effect. The requested operation is ignored.

Recovery. Check whether you are referring to the correct file.

```
*ERROR* Segment is inconsistent because its last DETACHSEG
failed to complete
```

Cause. You attempted to attach a segment whose contents are inconsistent. This can happen if, while detaching a private segment, TACL begins to write the segment out but fails for any reason to write the entire file.

Effect. The requested operation is ignored.

Recovery. Re-create the segment file.

```
*ERROR* Segment version incompatible
```

Cause. You attempted to attach a segment file that was last changed by a newer RVU of TACL, and the newer RVU has implemented incompatible changes in the segment file format.

Effect. The requested operation is ignored.

Recovery. Re-create the segment file.

```
*ERROR* SPI buffer does not begin with -28
```

Cause. In an #SSxxx function (other than #SSINIT), you supplied a buffer whose first16 bits do not contain the value -28.

Effect. The requested operation is ignored.

Recovery. Ensure that each SPI buffer structure begins with -28 in its first 16 bits.

```
*ERROR* Specified character class not present in CPRULES file
```

Cause. Indeterminate.

Effect. The requested operation is ignored.

Recovery. See your service provider to obtain a new copy of the CPRULES file in question.

```
*ERROR* Stack overflow
```

Cause. TACL experienced a processor stack overflow, possibly because of a very large structure or I/O buffer.

Effect. The requested operation is ignored.

Recovery. Reexamine your coding to find ways in which structures or I/O buffers can be made smaller.

```
*ERROR* STRUCT is not long enough to be nulled by given token
map
```

Cause. In an #SSxxx function, you supplied a token map and a structure whose data area was smaller than that described by the token map.

Effect. The requested operation is ignored.

Recovery. Ensure that the structure involved has enough data area to accommodate the token map's specifications

```
*ERROR* STRUCT's data area is not long enough
```

Cause. You attempted to use a structure as an SPI buffer, but the structure is either too short to hold a header or is shorter than the header claims.

Effect. The requested operation is ignored.

Recovery. Ensure that the structure has a data area large enough to accommodate the header and its requirements.

```
*ERROR* STRUCT's data would exceed 5000 bytes
```

Cause. You attempted to define a structure whose data area would be longer than the TACL limit of 5000 bytes per structure.

Effect. The requested operation is ignored.

Recovery. Check and adjust the individual item definitions in the STRUCT to prevent their aggregate from exceeding the maximum structure size

```
*ERROR* SUPER.SUPER may not LOGON remotely
```

Cause. The system manager of the remote system where you started your TACL process has configured that TACL so that the super ID cannot log onto it if it is started remotely from a remote system.

Effect. The LOGON operation is ignored.

Recovery. None. (You can, however, log on to the remote system under an allowable ID, start a TACL process locally on that system, and then log on as the super ID.)

```
*ERROR* Syntax error in lower bound
```

Cause. You attempted to define the lower end of a range in a structure and failed to supply a number, or the number was not terminated by a colon or closing parenthesis.

Effect. The requested operation is ignored.

Recovery. Correct the range specification.

```
*ERROR* Syntax error in upper bound
```

Cause. You attempted to define the upper end of a range in a structure and failed to supply a number, or the number was not terminated by a closing parenthesis.

Effect. The requested operation is ignored.

Recovery. Correct the range specification.

```
*ERROR* System not available
```

Cause. Either all communication paths to the specified system are down, or the system itself is down.

Effect. The requested operation is ignored.

Recovery. Wait until the system is available.

```
*ERROR* System's base address for selectable segments has
been changed.
```

Cause. The system address for selectable segments has changed.

Effect. The requested operation is ignored.

Recovery. Contact your service provider.

```
*ERROR* TACL internal buffer too small
```

Cause. An internal buffer was exceeded.

Effect. The requested operation is ignored.

Recovery. Decrease the size of macros or routines where possible, avoid excessive nesting of macros or routines, and structure recursive macros so that they invoke themselves in their last statements, so that each instance of the macro vanishes from the text buffer before the next one begins. Use loops to perform operations on large amounts of data a little at a time. Use options, such as LOADED in the LOAD command, or other similar means, to direct results to variables. Use the VOLUME command to reduce the number of template fields you need to pass to #FILENAMES.

```
*ERROR* TACL not named, cannot have a backup
```

Cause. You attempted to define a backup processor with a (#)BACKUPCPU operation, but your TACL has no process name.

Effect. The requested operation is ignored.

Recovery. Start over with a TACL that is a named process.

```
*ERROR* TACL process must be named
```

Cause. You attempted to use a server from an unnamed TACL process.

Effect. The requested operation is ignored.

Recovery. Certain operations, including #SERVER and the INV, OUTV, and STATUS options of the RUN command, require that TACL be a named process. Restart the TACL process with a name.

```
*ERROR* TACL stopped by a process ABEND/STOP (pid: nn, nnn)
```

Cause. You specified STOPONABEND as a parameter to the TACL command, and a process it started abended or stopped with completion code -1, -2, -3, 2, 3, 4, 5, or 6.

Effect. The TACL that started the abended or stopped process also stops.

Recovery. Determine why the process abended or stopped, correct the problem, and restart your TACL process.

```
*ERROR* Text buffer overflow
```

Cause. You attempted to perform an operation that exceeded the 30,000-byte capacity of the text buffer, which is used to hold TACL statements while they are being processed (text is removed from the text buffer when TACL executes it).

Each line break uses one byte; each square bracket, vertical bar, or “~_” space uses two bytes; other characters use one byte each. Comments of the { text } and == text forms are omitted, and trailing spaces are nearly always omitted. Other effects on the text buffer are:

- The TACL IN file is read into the text buffer until any square brackets have been balanced.
- File or variable macros, when invoked, are read entirely into the text buffer; dummy arguments are substituted as they are read in.
- All other variable types, when invoked, are read entirely into the text buffer.
- During execution, the space available in the text buffer diminishes as routines and macros are nested. Also, TACL can use up to half of the available area of the text buffer as a scratch area.

Common causes of “Text buffer overflow” are:

- Very large macros or routines. As a rule of thumb, macros should never exceed15,000 bytes after dummy argument substitution; routines should never exceed15,000 bytes. Nesting reduces these numbers further.
- Uncontrolled nesting of macros. (If, however, you construct a recursive macro so that it invokes itself in its last statement, each instance of the macro vanishes from the text buffer before the next one begins, and it does not overflow the text buffer).

  Routine nesting, on the other hand, more commonly runs out of routine stack space (“Routine stack overflow”) instead of text buffer space.
- Matching too many file names with #FILENAMES

Effect. The requested operation is ignored.

Recovery. Examine your TACL program to look for ways to avoid text buffer overflow:

Decrease the size of macros or routines where possible, avoid excessive nesting of macros or routines, and structure recursive macros as described above. Use loops to perform operations on large amounts of data a little at a time. Use options, such as LOADED in the LOAD command, or other similar means, to direct results to variables instead of to the text buffer. Use the VOLUME command to reduce the number of template fields you need to pass to #FILENAMES.

```
*ERROR* This type of variable is inappropriate here
```

Cause. You indicated an inappropriate type for the variable; for example, it would be inappropriate to use the DELTA type for any variable other than one that contains #DELTA commands.

Effect. The requested operation is ignored.

Recovery. Correct the variable type.

```
*ERROR* Token map contains a specification unknown to TACL
```

Cause. You supplied a token map with a format that TACL does not understand. The token map is probably invalid, corrupted, or includes some new feature unknown to this RVU of TACL.

Effect. The requested operation is ignored.

Recovery. Examine the token map and repair as the situation indicates.

```
*ERROR* Token map would overflow its STRUCT
```

Cause. In an #SSxxx function, you supplied a token map and a structure whose data area was smaller than that described by the token map.

Effect. The requested operation is ignored.

Recovery. Ensure that the STRUCT has a large enough data area to accommodate the requirements of the token map.

```
*ERROR* Too long
```

Cause. The indicated text is too long for the context in which it is being used.

Effect. The requested operation is ignored.

Recovery. Shorten the text.

```
*ERROR* Too many arguments
```

Cause. You entered too many arguments for the request; for example, SETMANY allows not more than 100 variables.

Effect. The requested operation is ignored.

Recovery. Delete the excess arguments.

```
*ERROR* Too many options
```

Cause. You specified more options than are allowed.

Effect. The requested operation is ignored.

Recovery. Delete the additional options and reissue the request

```
*ERROR* Too many PARAMs
```

Cause. You specified more PARAMs than are allowed.

Effect. The requested operation is ignored.

Recovery. Your PARAM assignments must not exceed 1024 bytes of internal storage

```
*ERROR* Too many simultaneous exceptions
```

Cause. Too many exceptions have been filtered.

Effect. The requested operation is ignored.

Recovery. You are allowed a cumulative total of 32 different exceptions.

```
*ERROR* Too many successive aliases or alias loop
```

Cause. You attempted to invoke more than 16 successive aliases, which are referring to each other in an endless chain.

Effect. The requested operation is ignored.

Recovery. Reduce the number of successive aliases so there are no more than 16.

Note . Each of these trap-related messages is followed by a line showing the register contents at the time the trap occurred:

```
Trap information: S = % nnnnn, P = % nnnnnn, CS = % nnn,
E = % nnnnnn, L = % nnnnnn
```

S, P, E, and L are the TACL registers; CS is the code segment for the P register.

Additional information about trap errors and trap handling is given in the *Guardian Programmer's Guide* and the *Guardian Procedure Calls Reference Manual*.

```
*ERROR* TRAP 0 in TACL: Illegal address reference
```

Cause. An address was specified that was not within either the virtual code area or the virtual data area allocated to the TACL process.

Effect. Indeterminate. Depending on context, the trap handler either recovers or abends the TACL process.

Recovery. None. Notify your service provider.

```
*ERROR* TRAP 1 in TACL: Instruction failure
```

Cause. An attempt was made to execute a code word that is not an instruction, or an illegal extended address reference was made.

Effect. Indeterminate. Depending on context, the trap handler either recovers or abends the TACL process.

Recovery. None. Notify your service provider.

```
*ERROR* TRAP 2 in TACL: Arithmetic overflow
```

Cause. Either the result of a signed arithmetic operation could not be represented with the number of bits available for the particular data type, or division by zero was attempted.

Effect. Indeterminate. Depending on context, the trap handler either recovers or abends the TACL process.

Recovery. None. Notify your service provider.

```
*ERROR* TRAP 3 in TACL: Stack overflow
```

Cause. An attempt was made to execute a procedure or subprocedure whose (sub)local data area extends into the upper 32KB of the data area. Stack overflow can also occur when calling a system procedure without sufficient remaining virtual data space for that procedure to run (the procedure is not executed).

Effect. Indeterminate. Depending on context, the trap handler either recovers or abends the TACL process.

Recovery. None. Notify your service provider.

```
*ERROR* TRAP 4 in TACL: Process loop timer timeout
```

Cause. TACL has issued a call to the SETLOOPTIMER procedure and the time limit specified in the latest call has expired.

Effect. Indeterminate. Depending on context, the trap handler either recovers or abends the TACL process.

Recovery. None. Notify your service provider.

```
*ERROR* TRAP 11 in TACL: Memory manager disk read error
```

Cause. An unrecoverable read error occurred while attempting to bring in a page from virtual memory.

Effect. Indeterminate. Depending on context, the trap handler either recovers or abends the TACL process.

Recovery. None. Notify your service provider.

```
TRAP 12 in TACL: No memory available or swap volume full
```

Cause. A page fault occurred, but no physical memory page is available for overlay, or the swap volume is full, preventing the memory manager from saving the memory page.

Effect. Indeterminate. Depending on context, the trap handler either recovers or abends the TACL process.

Recovery. Delete some swap volume contents to make room for TACL usage.

```
*ERROR* TRAP 13 in TACL: Uncorrectable memory error
```

Cause. An uncorrectable memory error was detected.

Effect. Indeterminate. Depending on context, the trap handler either recovers or abends the TACL process.

Recovery. None. Notify your service provider.

```
*ERROR* Unable to access userid file
```

Cause. TACL was unable to access the system USERID file.

Effect. The requested operation is ignored.

Recovery. If you are the super ID (user 255,255), check the status of the user ID file and take appropriate action. If you are not the super ID, you cannot access the user ID file.

```
*ERROR* Unable to allocate contiguous word space
(Hint: You probably should #LOGOFF and then LOGON)
```

Cause. There was not enough contiguous word space available.

Effect. The requested operation is ignored.

Recovery. Use #LOGOFF, then log on again.

```
*ERROR* Unable to allocate map
```

Cause. When attempting to run a program, there was not enough space in the MAPPOOL of the processor.

Effect. The requested operation is ignored.

Recovery. Run the program on another CPU, or wait until there is room on the chosen CPU.

```
*ERROR* Unable to allocate space in segment: seg-file-name
(HINT: You probably should #LOGOFF/SEGRELEASE/)
```

Cause. Your use of variables has consumed all of the 1024 pages that can be allocated to one private segment.

Effect. The requested operation is ignored.

Recovery.

- If TACL is interactive, use #LOGOFF/SEGRELEASE/, then log on again. Analyze your usage and either reduce it or distribute it among multiple private segments.
- If TACL is noninteractive, TACL abends with an abnormal completion code.

```
*ERROR* Unable to communicate with system monitor process:
text
```

Cause. When you attempted to run a program, TACL could not communicate with the system monitor process in the requested CPU, possibly because the CPU did not exist or was down.

Effect. The requested operation is ignored.

Recovery. Try running the program on another CPU.

```
*ERROR* Unable to expand macro
```

Cause. TACL could not return the result of the indicated macro because a syntax error occurred in a dummy argument.

Effect. The requested operation is ignored.

Recovery. A prior message indicates the specific problem.

```
*ERROR* Unable to express default volume in network form
```

Cause. The default volume is not in the correct format for network usage.

Effect. The requested operation is ignored.

Recovery. For network access, device names, including names of disk volumes, can have no more than six characters (not including $).

```
*ERROR* Unable to express a filename in network form
```

Cause. The specified file name is not in the correct format for network usage.

Effect. The requested operation is ignored.

Recovery. Check that you have specified the file name correctly. For network access, device names, including names of disk volumes, can have no more than six characters. Process names can have no more than four characters.

```
*ERROR* Unable to get virtual disk space or error on swap
file: text
```

Cause. When attempting to run a program, there was not enough disk space or there was an error in the swap file for the user data memory.

Effect. The requested operation is ignored.

Recovery. Specify a different volume for the swap file.

```
*ERROR* Unable to load SECTION
```

Cause. TACL was unable to load the library file section.

Effect. The requested operation is ignored.

Recovery. A prior message indicates the specific problem.

```
*ERROR* Unable to obtain disk space for extent [ , File:
file-name ]
```

Cause. The file system attempted to add another extent to the file, but there was no available space on the disk.

Effect. The requested operation is ignored.

Recovery. Purge any unnecessary files and try the operation again. If it still fails, contact your system manager.

```
*ERROR* Unable to perform edit
```

Cause. #DELTA was unable to perform the requested edit.

Effect. The requested operation is ignored.

Recovery. A prior message indicates the specific problem.

```
*ERROR* Unknown delta command
```

Cause. The command you entered is not a valid #DELTA command.

Effect. The requested operation is ignored.

Recovery. A prior message indicates the specific problem.

```
*ERROR* Unknown function key
```

Cause. You pressed a function key (CTRL/A, for example) that is not recognized by TACL.

Effect. The requested operation is ignored.

Recovery. Reissue the function key, making sure that it is valid.

```
*ERROR* Unlicensed privileged program: text
```

Cause. You attempted to run a program that calls procedures it is not authorized to call. The super ID must license program files, by means of the FUP LICENSE command, to enable them to execute such procedures.

Effect. The requested operation is ignored.

Recovery. Check the name of the program. If you need to run the program, have the super ID license the program.

```
*ERROR* Unrecognised floating point type in object file
```

Cause. You used a floating-point format that is not recognized by the system.

Effect. The operating system does not allow the creation of the process.

Recovery. Use an appropriate floating-point format.

```
*ERROR* Unsupported CPRULES file version
```

Cause. The CPRULES file version is no longer supported by this RVU of TACL.

Effect. The requested operation is ignored.

Recovery. See your service provider to obtain a new copy of the CPRULES file in question.

```
*ERROR* Variable does not exist
```

Cause. You referred to a nonexistent variable. (This problem also occurs if you embed a dollar sign in a text argument that is terminated by a dollar sign.)

Effect. The requested operation is ignored.

Recovery. Check the spelling of the variable reference; if it is correct, ensure that you define the variable before referring to it. (Check to see if the argument can be bounded by alternative delimiters.)

```
*ERROR* Variable does not have that level
```

Cause. You referred to a nonexistent level of an existing variable.

Effect. The requested operation is ignored.

Recovery. Check the level specification, and check your method of creating levels for the variable, to ensure that they correspond.

```
*ERROR* Variable is in a shared segment making it read-only
```

Cause. You attempted to push, pop, change the value of, or set a breakpoint on a variable in a shared segment. The contents of shared segments cannot be changed. (Setting a breakpoint forces TACL to turn on a flag in the variable, thus changing it.)

Effect. The requested operation is ignored.

Recovery. First detach the segment, then reattach it in the private mode before attempting to change any variables in it. Return the segment to the shared mode afterward.

```
*ERROR* Variable is not numeric
```

Cause. A numeric variable is required.

Effect. The requested operation is ignored.

Recovery. Correct the indicated variable. A numeric variable can contain nothing other than a single decimal number, optionally preceded or followed by spaces and blank lines.

```
*ERROR* VT Internal error VTLAUNCH: at code location nnn
```

Cause. There is an internal problem within the VTLAUNCH procedure at code location *nnn*.

Effect. VTLAUNCH did not create the window or start the process.

Recovery. Report this error and code location to your service provider.

```
*ERROR* VT TELNET SCP window add failure TELNET: code
```

Cause. TELNET returned an SCP ADD error.

Effect. VTLAUNCH was unable to add a new TELNET window. The window was not created, and the x6530 emulator process was not started.

Recovery. Refer to TELNET documentation for recovery details and information about the TELNET error code.

```
*ERROR* VT Unable to clone x6530 emulator
Unix: code
```

Cause. The x6530 emulator was unable to make a clone of itself.

Effect. The x6530 emulator process for the virtual terminal window was not started.

Recovery. Refer to the appropriate UNIX reference manual for recovery details and information about the UNIX error code.

```
*ERROR* VT Unable to communicate with windowing comm process
GUARDIAN: file system error
```

Cause. VTLAUNCH was unable to communicate with the windowing system communications process.

Effect. The virtual terminal window was not created.

Recovery. Refer to the *Guardian Procedure Errors and Messages Manual* for recovery and file system error information.

```
*ERROR* VT Unable to open windowing comm process
GUARDIAN: file system error
```

Cause. VTLAUNCH was unable to open the windowing system communications process.

Effect. The virtual terminal window was not created.

Recovery. Refer to the *Guardian Procedure Errors and Messages Manual* for recovery and file system error information.

```
*ERROR* VT Unable to communicate with emulator
GUARDIAN: file system error
```

Cause. VTLAUNCH was unable to communicate with the emulator whose file number was specified.

Effect. The virtual terminal window was not created.

Recovery. Refer to the *Guardian Procedure Errors and Messages Manual* for recovery and file system error information.

```
*ERROR* VT Cannot determine proper window controller process
GUARDIAN: file system error
```

Cause. VTLAUNCH was unable to determine the proper window controller process with which to communicate.

Effect. The virtual terminal window was not created.

Recovery. Refer to the *Guardian Procedure Errors and Messages Manual* for recovery and file system error information.

```
*ERROR* VT Virtual Terminal Launch not supported by:
VTLAUNCH object, communication controller or emulator
```

Recovery. The virtual terminal window facility is not supported. One or more of the components (VTLNCH object, x6530 emulator, or TELNET) is missing or is the wrong version.

Cause. The virtual terminal window was not created.

Recovery. Ensure that the required version of each component is installed and running.

```
*ERROR* VT Illegal terminal name passed
GUARDIAN XBOUNDS: Xbounds-error
```

Cause. There is an internal problem in TACL or the operating system.

Effect. The window was not created and the process was not started.

Recovery. Report the message and error code to your service provider.

```
*ERROR* VT Unable to start the newly added TELNET window
TELNET: code
```

Cause. A window process was successfully added but could not be started.

Effect. The window was not created and the process was not started.

Recovery. Refer to TELNET documentation for recovery details and information about the TELNET error code.

```
*ERROR* Was not pushed
```

Cause. You attempted to set a variable that had not been pushed.

Effect. The requested operation is ignored.

Recovery. Push the variable and then set it.

```
*ERROR* Year must be four digits
```

Cause. You specified a year that does not contain 4 digits.

Effect. The command fails.

Recovery. Specify a 4-digit number for the year.

```
*ERROR* You already have a current INLINE process
```

Cause. You attempted to start a process using the INLINE option while an inline process already exists (#INLINEPROCESS is not empty).

Effect. The requested operation is ignored.

Recovery. Push #INLINEPROCESS before starting an inline process, or finish activities with the current inline process before starting another one.

```
*ERROR* You cannot explicitly set this variable
```

Cause. You attempted to assign a value to a built-in variable that can only receive a value as a side effect of some other operation. For example, you cannot explicitly set #INLINEPROCESS; it receives its value as the result of starting a process with the INLINE option, or terminating such a process.

Effect. The requested operation is ignored.

Recovery. None; the attempted operation is prohibited.

```
*ERROR* You have no INLINE process
```

Cause. You began an input line with the current INLINE prefix, or you issued an INLEOF command or #INLINEEOF invocation while no inline process exists.

Effect. The requested operation is ignored.

Recovery. Start a process with the INLINE option before attempting to use the INLINE feature.

```
*ERROR* ? line not allowed here
```

Cause. A line beginning with ? appeared in a place where such a line is not allowed.

Effect. The requested operation is ignored.

Recovery. Remove the line and retry the operation.

```
*WARNING* Could not access CPRULES0 file
text-reason
Using internal character processing rules tables
```

Cause. An error occurred while accessing the CPRULES0 file.

Effect. TACL continues initialization.

Recovery. Correct the error condition described in *text-reason.*

```
*WARNING* text buffer overflow, could not columnize the
variables.
```

Cause. For a large macro file there are too many variables to be columnized.

Effect. The variables are loaded but remain uncolumnized

Recovery. None.

```
*WARNING* VARIABLES okay, but text buffer overflowed. Could
not columnize variables.
```

Cause. For a large segment file there are too many variables to be columnized.

Effect. The variables are sorted but remain uncolumnized and the final result does not flag directory names with an asterisk (*).

Recovery. None.

DEFINE Error Messages

The following messages relate specifically to DEFINEs. Each (except the first, which is a TACL message) is associated with a numeric error code returned by the DEFINE-oriented procedure invoked. If one of these operating system procedures returns an error number that TACL does not recognize, TACL simply displays:

```
DEFINE error num
```

See the *Guardian Procedure Errors and Messages Manual* for additional details about these messages.

```
Attribute value too long for TACL
```

Cause. A DEFINE attribute value returned by the operating system is too long for the buffer TACL uses in obtaining such information. This can occur when the operating system reformats an attribute on output into a form different from that in which the attribute was input, such as when a value is input as a comma-separated list without spaces and the operating system inserts a space after each comma, causing the value to exceed the TACL limit of 1024 bytes.

Effect. The requested operation is ignored.

Recovery. Decrease the size of the original value to allow for reformatting by the operating system.

```
Current attribute is incomplete and inconsistent
```

Cause. The working set is invalid.

Effect. The operation is ignored.

Recovery. Correct the working set, then retry the operation.

```
Current attribute set is incomplete
```

Cause. A required attribute is missing from the current CLASS in the working set.

Effect. The operation is ignored.

Recovery. Add the attribute for the current CLASS, then retry the operation

```
Current attribute set is inconsistent, check number num
```

Cause. The working set is inconsistent for the current CLASS.

Effect. The operation is ignored.

Recovery. Correct the working set, then retry the operation. See Table 8-7 on page 8-185 for consistency rules.

```
DEFINE already exists " name"
```

Cause. You attempted to add a DEFINE name that already exists.

Effect. The operation is ignored.

Recovery. Correct or change the DEFINE name, then reissue the request.

```
DEFINE cannot be deleted " name"
```

Cause. You attempted to delete a DEFINE that cannot be deleted: for example, the =_DEFAULTS DEFINE.

Effect. The DEFINE is not deleted.

Recovery. None; the operation is not allowed.

```
DEFINE does not exist " name"
```

Cause. A DEFINE could not be found.

Effect. The operation is ignored.

Recovery. Check to be sure you are specifying the correct DEFINE name, then reissue the request.

```
DEFINE error 2054
```

Cause. There was a bounds error (in a parameter) for which TACL has no descriptive text.

Effect. The operation is ignored.

Recovery. This is a coding error; corrective action is application dependent.

```
DEFINE error 2056
```

Cause. An attribute was missing; TACL has no descriptive text to add.

Effect. The operation is ignored.

Recovery. Add the attribute, then retry the operation.

```
DEFINE error 2073
```

Cause. You attempted to replace the =_DEFAULTS DEFINE with a DEFINE having the same name but a class other than DEFAULTS.

Effect. The operation is ignored.

Recovery. Specify the DEFINE with a DEFAULTS class, then retry the operation.

```
DEFINE name not allowed under current defmode setting
```

Cause. The DEFMODE setting for the process does not allow the addition of a DEFINE type.

Effect. The operation is ignored.

Recovery. Change the DEFMODE setting to allow the desired operation.

```
Illegal attribute name " attr"
```

Cause. There was a syntax error in an attribute name.

Effect. The operation is ignored.

Recovery. Correct the syntax, then reissue the operation.

```
Illegal class name " name"
```

Cause. The CLASS name identified a nonexistent class.

Effect. The operation is ignored.

Recovery. Correct the CLASS name, then reissue the request.

```
Illegal DEFINE name " name"
```

Cause. There was a syntax error in the DEFINE name.

Effect. The requested operation is ignored.

Recovery. Correct the syntax error, then reissue the request.

```
Illegal value for " attr"
```

Cause. You supplied an invalid value for an attribute.

Effect. The operation is ignored.

Recovery. Correct the value, then reissue the request.

```
Required attribute - Cannot be reset
```

Cause. You attempted to reset a required DEFINE attribute.

Effect. The operation is ignored.

Recovery. None; you cannot reset a required DEFINE attribute.

```
Required parameter is not supplied
```

Cause. You failed to supply a required parameter.

Effect. The operation is ignored.

Recovery. Add the missing parameter, then reissue the request.

```
There is no attribute " attr" for the current class
```

Cause. You supplied an attribute that is not allowed for the current CLASS.

Effect. The operation is ignored.

Recovery. Correct the attribute for the current CLASS, then retry the operation.

```
Unable to obtain file system buffer space
```

Cause. Either file-system buffer space was not available or there is not enough room in the process file segment (PFS). For example, too many files are open or too many no-wait I/O operations are outstanding.

Effect. The operation is ignored.

Recovery. Close some files, wait for no-wait I/O to finish, then try again. Check the system for processes that use too much buffer space. If the problem persists, call your service provider.

```
Unable to obtain physical memory
```

Cause. There was not enough memory available to perform the requested operation.

Effect. The operation is ignored.

Recovery. Wait, then try again. If the problem persists, check the system for processes that use too much memory.

Process Creation Error Messages

If you receive a message indicating that the program file or library file has an illegal format (*ERROR* NEWPROCESS error 006, *nnn*), TACL follows that message with an error that describes why the file is unacceptable. Following is a partial list of errors:

```
*ERROR* Not a disk file
*ERROR* Not file code 100 or 700
*ERROR* Not correct file structure
*ERROR* Requires later version of GUARDIAN
*ERROR* No main procedure
*ERROR* LIB file has main procedure
*ERROR* No data pages
*ERROR* Invalid PEP
*ERROR* Header INITSEGS not consistent with size
*ERROR* Resident size greater than code area
*ERROR* File not fixed-up by binder
*ERROR* File has undefined data blocks
*ERROR* Unresolved references from data block to code block
*ERROR* Too many code spaces
*ERROR* Bad file or Bad target
```

Note. "*ERROR*BadfileorBadtarget"occurswhentryingtoruna700codefileonaTNS/Esystem remotely from a TNS/R system.

See the *Guardian Procedure Errors and Messages Manual* for a description of the subcode, and of process creation error messages in general.

If the error is not in a defined category, TACL displays *ERROR* Subcode nnn.

The #ERRORNUMBERS built-in variable stores four space-separated numbers after a process creation operation. The first of the four numbers returned, 1101, represents the process creation error (if no error occurred, all four numbers are zero). The second number identifies the specific error, as listed in Table B-1.

Table B-1. #ERRORNUMBERS Results (page 1 of 2)

Number	Error
1	Process has undefined externals (warning only; process was started)
2	No process control block available
3	File-system error occurred on program file
4	Unable to allocate map
5	File-system error occurred on swap file
6	Illegal file format
7	Unlicensed privileged program
8	Process name error
9	Library conflict
10	Unable to communicate with system monitor process

Table B-1. #ERRORNUMBERS Results (page 2 of 2)

Number	Error
11	File-system error occurred on library file
12	Program file and library file specified are the same file
13	Extended data segment initialization error
14	Extended segment swap file error
15	Illegal home terminal

If the second number is 3, 5, 8, 11, 13, 14, or 15, the third number contains a file-system error number identifying the specific error condition.

If the second number is 6, the third number contains either 0, specifying that the program file is in error, or 1, specifying that the library file is in error. The fourth number contains a value in the range 1-14 that specifies why the file format is invalid. The numbers correspond to the positions in the list of process creation “illegal format” error messages given in the preceding list; for example, 3 indicates the main procedure is missing.

RCVDUMP Error Messages

The following messages are returned by the RCVDUMP procedure. Use of the RCVDUMP and RECEIVEDUMP commands is restricted to super-group users only.

RCVDUMP error messages are divided into two subsections:

- Messages for H-series only. These are all designated by the prefix ‘DUMP_’.
- Messages common to all systems, H-series, G-series and D-series. The wording may differ slightly between systems.

RCVDUMP Error Messages for H-Series Only

```
DUMP _SLICE_NOT_SPECIFIED
```

Cause. You specified a parallel dump without the slice option, but more than one slice is in a stopped state.

Effect. RCVDUMP does not continue processing.

Recovery. Specify the slice from which the PE is to be dumped.

```
DUMP_PARAM_CONFLICT
```

Cause. You specified both parallel and online options. This is not possible.

Effect. RCVDUMP does not continue processing.

Recovery. Redo the command with only one of these options.

```
DUMP_INSUFFICIENT_PRIVILEGE
```

Cause. Receive dump was not started by super user id. Your current id cannot perform this operation.

Effect. RCVDUMP does not continue processing.

Recovery. Log in as the super user.

```
DUMP_NOT_LOCAL
```

Cause. You cannot dump a remote system.

Effect. RCVDUMP does not continue processing.

Recovery. Run RCVDUMP on the remote system.

```
DUMP_SLICE_NOT_STOPPED
```

Cause. The PE on the slice you specified has not been stopped.

Effect. RCVDUMP does not continue processing.

Recovery. You must stop that PE or specify an alternative.

```
DUMP_CPU_NOT_HALTED
```

Cause. The logical cpu has not halted.

Effect. RCVDUMP does not continue processing.

Recovery. Either halt that cpu or specify another one.

```
DUMP_NO_DISK_SPACE
```

Cause. There is not enough space on the disk that contains your designated dump file.

Effect. RCVDUMP does not continue processing.

Recovery. Select a dump file on another disk.

```
DUMP_PRIME_FAILED
```

Cause. Priming the CPU for an HSS dump failed.

Effect. RCVDUMP does not continue processing.

Recovery. Contact your HP service provider.

```
DUMP_NO_MEMORY
```

Cause. Couldn't allocate memory for internal data buffers.

Effect. RCVDUMP does not continue processing.

Recovery. Contact your HP service provider.

```
DUMP_COMM_FAILURE
```

Cause. RCVDUMP couldn't talk to the downed CPU.

Effect. RCVDUMP does not continue processing.

Recovery. Contact your HP service provider.

```
DUMP_MEM_CONFIG_ERROR
```

Cause. Memory configuration check failed.

Effect. RCVDUMP does not continue processing.

Recovery. Contact your HP service provider.

```
DUMP_COMPRESSED_DATA_ERROR
```

Cause. Compressed data received from HSS is not correct.

Effect. RCVDUMP does not continue processing.

Recovery. Contact your HP service provider.

```
DUMP_INVALID_STARTER_PKT
```

Cause. Receive dump has received invalid starter packet from HSS.

Effect. RCVDUMP does not continue processing.

Recovery. Contact your HP service provider.

```
DUMP_PROCESS_CREATE_ERROR
```

Cause. Online dump start failed.

Effect. RCVDUMP does not continue processing.

Recovery. Contact your HP service provider.

```
DUMP_ONLINE_PROCESS_ABEND
```

Cause. Online dump process abended.

Effect. RCVDUMP does not continue processing.

Recovery. Contact your HP service provider.

```
DUMP_DUPLICATE_INVOCATION
```

Cause. Already one instance of receive dump is running on this logical CPU.

Effect. RCVDUMP does not continue processing.

Recovery. Contact your HP service provider.

```
DUMP_INTERNAL_ERROR
```

Cause. Internal error like disk file open failed or write to the disk file failed with any error other than no disk space.

Effect. RCVDUMP does not continue processing.

Recovery. Contact your HP service provider.

RCVDUMP Error Messages for H-Series, G-Series and D-Series

```
CPU nn HAS BEEN DUMPED TO file-name.
```

Cause. The processor has been successfully dumped to the specified file.

Effect. This is an informative message.

Recovery. Informative message only; no corrective action needed.

```
CPU nn HAS NOT BEEN RESET-LOAD.
```

Cause. You attempted to dump a processor that had not been through the reset-load sequence.

Effect. No processor is dumped.

Recovery. Do the reset-load and then retry the command.

```
CPU nn IS RUNNING, CANNOT BE DUMPED.
```

Cause. You attempted to dump a processor that is running.

Effect. No processor is dumped.

Recovery. Specify the number of a processor that is not running.

```
CPU nn MAY HAVE BEEN PARTIALLY DUMPED TO
file-name. IF THE LIGHTS OF THE DUMPED
CPU ARE NOT %177777, THEN PLEASE REDO THE
RESET-LOAD AND RECEIVEDUMP SEQUENCE.
```

Cause. A partial dump may have been taken of the specified processor.

Effect. The CPU may or may not have been dumped, as shown in the displayed lights.

Recovery. Check the lights. If they are %177777, the dump was taken successfully; if they are not %177777, reset-load the processor and reenter the command.

```
INVALID dump-file-name.
```

Cause. You specified an invalid file name.

Effect. No processor is dumped.

Recovery. Specify a valid file name.

```
INVALID cpu NUMBER, THE ALLOWED VALUES ARE 0: nn.
```

Cause. You specified an invalid CPU number.

Effect. No processor is dumped.

Recovery. Specify a valid CPU number

```
INVALID PARAMETER, THE ALLOWED VALUES ARE
[ PRIME NOPRIME ].
```

Cause. You specified something else when only PRIME or NOPRIME was expected.

Effect. No processor is dumped.

Recovery. Reenter the command specifying PRIME or NOPRIME.

```
INVALID bus id, THE ALLOWED VALUES ARE [ X Y ].
```

Cause. You specified an invalid bus.

Effect. No processor is dumped.

Recovery. Specify a valid bus.

```
IS NOT A DISK FILE.
```

Cause. You attempted to dump a processor to a nondisk file.

Effect. No processor is dumped.

Recovery. Retry the operation specifying a disk file

```
file-name IS NOT EMPTY.
```

Cause. You attempted to dump a processor to a nonempty disk file.

Effect. No processor is dumped.

Recovery. Either purge the old file or specify a different file, then reenter the command.

```
file-name IS NOT FILE CODE nnnn.
```

Cause. You attempted to dump a processor to an existing file whose file code was not that of a dump file.

Effect. No processor is dumped.

Recovery. Purge the old file or specify a file with the correct file code.

```
INVALID cpu NUMBER, THE ALLOWED VALUES ARE 0: nn.
```

Cause. You specified an invalid CPU number.

Effect. No processor is dumped.

Recovery. Specify a valid CPU number.

```
INVALID dump-file-name.
```

Cause. You specified an invalid file name.

Effect. No processor is dumped.

Recovery. Specify a valid file name.

```
MISSING bus id.
```

Cause. You did not specify the bus through which to dump the processor.

Effect. No processor is dumped.

Recovery. Specify a bus

```
MISSING dump-file-name.
```

Cause. You did not specify a dump file name.

Effect. No processor is dumped.

Recovery. Specify a file in the command.

```
MISSING cpu NUMBER.
```

Cause. You did not specify a processor to dump.

Effect. No processor is dumped.

Recovery. Specify a CPU number in the command.

```
UNABLE TO CREATE file-name, ERROR nnnn.
```

Cause. The specified file could not be created because file-system error *nnnn* occurred.

Effect. No processor is dumped.

Recovery. Correct the cause of the file-system error and reenter the command.

```
UNABLE TO OPEN file-name, ERROR nnnn.
```

Cause. The specified file could not be opened because file-system error *nnnn* occurred.

Effect. No processor is dumped.

Recovery. Correct the cause of the file-system error and reenter the command.

```
UNABLE TO PURGE file-name, ERROR nnnn.
```

Cause. The specified file could not be purged because file-system error *nnnn* occurred

Effect. No processor is dumped.

Recovery. Correct the cause of the file-system error and reenter the command.

```
UNABLE TO WRITE TO file-name, ERROR nnnn.
```

Cause. The specified file could not be written to because file-system error *nnnn* occurred.

Effect. No processor is dumped.

Recovery. Correct the cause of the file-system error and reenter the command.

```
** WARNING ** PRIME/NOPRIME IS NOT VALID FOR

THIS PROCESSOR TYPE: IGNORED.
```

Cause. You specified PRIME or NOPRIME and this feature is not supported for the type of processor.

Effect. No processor is dumped.

Recovery. Reenter the command without PRIME or NOPRIME.

RELOAD Error Messages

The following messages are returned by the RELOAD procedure. Use of the RELOAD command is restricted to super-group users only.

RELOAD error messages are divided into two subsections:

- Messages for H-series only
- Messages common to all systems, H-series, G-series and D-series. The wording may differ slightly between systems.

RELOAD Error Messages for H-Series Only

RELOAD prints several error messages at different stages. Many of these give reasons as to why the RELOAD of a particular CPU failed. The RELOAD process will still continue, since failure for one CPU does not indicate that other CPUs won't RELOAD. There are other error messages which indicate general problems (like file system errors) that are more likely in the initial stages of RELOAD. Such errors would

usually prevent RELOAD from going any farther. A third class of messages are warnings, which don't indicate the failure of RELOAD of any CPU, but indicate something amiss in the system.

```
Alternate OS FileSet name should be specified as :
$VOLUME.SYSnn.OSDIR or $VOLUME or SYSnn.OSDIR or OSDIR
```

Cause. RELOAD is trying to open the alternate file specified on the command line.

Effect. This is a command-line error. RELOAD will not continue processing that CPU specification.

Recovery. Make sure the alternate osdir file, its volume and its subvolume are properly specified in the command line.

```
String Too Long: <File name>
```

Cause. RELOAD is trying to open the alternate file specified on the command line.

Effect. This is a command-line error. RELOAD will not continue processing that CPU specification.

Recovery. Make sure the alternate osdir file, its volume and its subvolume are properly specified in the command line.

```
Remote alternate files not supported: <File name>
```

Cause. RELOAD is trying to open the alternate file specified on the command line.

Effect. This is a command-line error. RELOAD will not continue processing that CPU specification.

Recovery. Make sure the alternate osdir file, its volume and its subvolume are properly specified in the command line.

```
Multiple alternate files not supported: <File name>
```

Cause. RELOAD is trying to open the alternate file specified on the command line.

Effect. This is a command-line error. RELOAD will not continue processing that CPU specification.

Recovery. Make sure the alternate osdir file, its volume and its subvolume are properly specified in the command line.

```
Unable to FILENAME_EDIT_ <File name>
```

Cause. RELOAD is trying to FILENAME_EDIT_ the alternate filename specified on the command line.

Effect. This is a command-line error. RELOAD will not continue processing that CPU specification.

Recovery. Make sure the alternate osdir file, its volume and its subvolume are properly specified in the command line.

```
Unable to FILENAME_DECOMPOSE_ <File name>
```

Cause. RELOAD is trying to FILENAME_DECOMPOSE_ the alternate filename specified on the command line.

Effect. This is a command-line error. RELOAD will not continue processing that CPU specification.

Recovery. Make sure the alternate osdir file, its volume and its subvolume are properly specified in the command line.

```
Invalid OMITSLICE argument: <user specified slice>.
```

Cause. RELOAD is parsing the OMITSLICE parameter.

Effect. This is a command-line error. RELOAD will not continue processing that CPU specification.

Recovery. You must type on the command line one of A, B, or C, or no argument.

```
Repeated OMITSLICE option: <command line text>.
```

Cause. RELOAD is parsing the command line options.

Effect. This is a command-line error. RELOAD will not continue processing that CPU specification.

Recovery. You must type only one instance of OMITSLICE per CPU specification.

```
Alternate OS FileSet name should be specified as :
$VOLUME.SYSnn.OSDIR or $VOLUME or SYSnn.OSDIR or OSDIR
```

Cause. RELOAD is trying to open the alternate file specified on the command line.

Effect. This error occurs while opening or reading from the OS Fileset. RELOAD will not continue processing that CPU specification.

Recovery. Make sure the alternate osdir file, its volume and its subvolume are properly specified in the command line. If they are, see if the file is open, unreadable or of the wrong type.

```
Unable to open <file name>, file-system error: #<error
number>
```

Cause. RELOAD is trying to open one of the OS Fileset files and gets an error. The possible reasons include an internal error in RELOAD itself or a corrupted OS Fileset.

Effect. RELOAD does not continue processing.

Recovery. Contact your HP service representative.

Omitslice Information and Error Messages

TFDS is the usual method for using the omitslice option, but if you use a TACL command line method of controlling RELOAD you may encounter some of the following messages.

```
Warning: In reintegration. Retrying in one minute.
(CPU X, SLICE X)
or
Warning: Slice still in reintegration.
```

Cause. These two informational messages tell you to wait while the process is executing.

Effect. No effect. You must continue awaiting the outcome.

Recovery. No recovery. You must continue awaiting the outcome.

```
Failed to omit slice. Continuing with Reload of CPU. (CPU X)
or
Failed to omit slice. Aborting Reload of CPU. (CPU X)
```

Cause. These two error messages tell you what happened in the case of an unsuccessful action. There are several other variations of these messages depending on how far the RELOAD process got in execution.

Effect. The message text will tell you about the effect.

Recovery. You must decide whether to repeat the operation with different parameters, or contact your HP service representative

RELOAD Error Messages for H-Series, G-Series and D-Series

```
XXXXXX: alternate osimage file error code nnn.
```

Cause. There was an error (nnn) with the alternate system-image file.

Effect. What you typed (*XXXXXX*) is returned in uppercase along with the error.

Recovery. Specify a different file or make the desired file accessible.

```
XXXXXX: alternate osimage file is incompatible.
```

Cause. The alternate system-image file was not compatible.

Effect. What you typed (*XXXXXX*) is returned in uppercase along with the error.

Recovery. Specify a different file.

```
CPU nn already specified.
```

Cause. The indicated CPU was already specified earlier in the RELOAD command.

Effect. The CPU is reloaded only once.

Recovery. Correct and reissue the command.

```
CPU nn is already up.
```

Cause. An attempt was made to reload the indicated CPU, which was already up.

Effect. CPU *nn* is not reloaded.

Recovery. Correct the CPU number and reissue the command.

```
CPU nn is illegal.
```

Cause. An invalid CPU number was specified.

Effect. The RELOAD operation aborts.

Recovery. Correct the CPU number and reissue the command.

```
CPU nn is not configured.
```

Cause. An attempt was made to reload the indicated CPU, but the CPU was not configured at system generation time.

Effect. CPU *nn* is not reloaded.

Recovery. Correct the CPU number and reissue the command.

```
CPU nn reload FAILED: bad cpu number [, step # nn].
```

Cause. A bad CPU number was indicated. The optionally supplied step # *nn* indicates the execution step where the problem occurred:

01 to 09	Initialization and sending initial boots (microcode and bootstraps)
20 to 29	Common logic for sending packets over the bus (either initial boots or CPU image)
30 to 39	Setting breakpoints
80 to 99	Starting processing in the reloaded CPU, including sending the destination control table (DCT) starting processes, TMF initialization, setting the clock, and so on.

Effect. The reload of the indicated CPU fails.

Recovery. Correct the CPU number and reissue the command.

```
CPU nn reload FAILED: bad file type [, step # nn].
```

Cause. A bad system-image file was indicated. The optionally supplied step # *nn* indicates the execution step where the problem occurred:

01 to 09 Initialization and sending initial boots (microcode and bootstraps)

20 to 29 Common logic for sending packets over the bus (either initial boots or CPU image)

30 to 39 Setting breakpoints

80 to 99 Starting processing in the reloaded CPU, including sending the destination control table (DCT) starting processes, TMF initialization, setting the clock, and so on.

Effect. The reload of the indicated CPU fails.

Recovery. Check that you used the correct alternate system-image file. Correct it or try a different one, and reissue the command.

```
CPU nn reload FAILED: bus or loading error [, step nn].
```

Cause. The attempted reload was aborted because of a bus or loading error, or the default bus is down. There may have been a failure of another CPU during the reload.

The optionally supplied step # *nn* indicates the execution step where the problem occurred:

01 to 09 Initialization and sending initial boots (microcode and bootstraps)

20 to 29 Common logic for sending packets over the bus (either initial boots or CPU image)

30 to 39 Setting breakpoints

80 to 99 Starting processing in the reloaded CPU, including sending the destination control table (DCT) starting processes, TMF initialization, setting the clock, and so on.

Effect. The reload of the indicated CPU fails.

Recovery. Prime the CPU and retry the command. If this does not work, the CPU or bus could be bad; try another bus.

```
CPU nn reload FAILED: File system error # nnn [, step # nn].
```

Cause. There was a file-system error, probably with the system-image file, on the attempted reload. The optionally supplied step # *nn* indicates the execution step where the problem occurred:

01 to 09 Initialization and sending initial boots (microcode and bootstraps)

20 to 29 Common logic for sending packets over the bus (either initial boots or CPU image)

30 to 39 Setting breakpoints

80 to 99 Starting processing in the reloaded CPU, including sending the destination control table (DCT) starting processes, TMF initialization, setting the clock, and so on.

Effect. The reload of the indicated CPU fails.

Recovery. See “File-System Errors” in the *Guardian Procedure Errors and Messages Manual* for corrective action for the error indicated by *nnn*. If this does not correct the problem, reprime the CPU and run RELOAD in a different CPU.

```
CPU nn reload FAILED: LCB allocation failed [, step # nn].
```

Cause. There were not enough LCBs to handle the reload. The optionally supplied step # *nn* indicates the execution step where the problem occurred:

01 to 09 Initialization and sending initial boots (microcode and bootstraps)

20 to 29 Common logic for sending packets over the bus (initial boots or CPU image)

30 to 39 Setting breakpoints

80 to 99 Starting processing in the reloaded CPU, including sending the destination control table (DCT) starting processes, TMF initialization, setting the clock, and so on.

Effect. The reload of the indicated CPU fails.

Recovery. Retry the command. If that does not work, use PEEK to determine which CPUs have LCBs available and run RELOAD in a different CPU.

```
CPU nn reload FAILED: simultaneous RELOAD attempted [, step #
nn].
```

Cause. An attempt was made to do more than one reload at the same time. The optionally supplied step # *nn* indicates the execution step where the problem occurred:

01 to 09 Initialization and sending initial boots (microcode and bootstraps)

20 to 29 Common logic for sending packets over the bus (initial boots or CPU image)

30 to 39 Setting breakpoints

80 to 99 Starting processing in the reloaded CPU, including sending the destination control table (DCT) starting processes, TMF initialization, setting the clock, and so on.

Effect. The reload of the indicated CPU fails.

Recovery. Coordinate your reload with anyone else who is also attempting a reload.

```
CPU nn reloaded successfully.
```

Cause. The attempted reload of the indicated CPU was successful.

Effect. The indicated CPU is reloaded.

Recovery. Informative message; no corrective action needed.

```
duplicate osimage filename.
```

Cause. The same system-image file name was indicated more than once.

Effect. What you typed (XXXXXX) is returned in uppercase along with the error.

Recovery. Correct the file name for the system-image and reissue the command.

```
XXXXXX: ignored
```

Cause. The indicated command was ignored.

Effect. What you typed (*XXXXXX*) is returned in uppercase along with the error.

Recovery. Correct and reissue the command.

```
XXXXXX: invalid bus.
```

Cause. An invalid bus was indicated.

Effect. The RELOAD command aborts. What you typed (*XXXXXX*) is returned in uppercase along with the error.

Recovery. Correct and reissue the command.

```
XXXXXX: invalid cpu.
```

Cause. An invalid CPU was indicated.

Effect. What you typed (*XXXXXX*) is returned in uppercase along with the error.

Recovery. Correct and reissue the command.

```
Invalid cpu list.
```

Cause. The list of CPUs supplied to RELOAD was unacceptable.

Effect. The RELOAD command aborts.

Recovery. Correct and reissue the command.

```
XXXXXX: invalid option.
```

Cause. An invalid option was specified.

Effect. What you typed (*XXXXXX*) is returned in uppercase along with the error.

Recovery. Valid options are NOSWITCH, bus, volume, and file-name. Correct the option and reissue the command.

```
Invalid range.
```

Cause. The indicated CPU range was unacceptable.

Effect. The RELOAD command aborts.

Recovery. Correct and reissue the command.

```
XXXXXX: is not a recognized command.
```

Cause. The indicated command is not recognized by RELOAD.

Effect. What you typed (*XXXXXX*) is returned in uppercase along with the error.

Recovery. Correct and reissue the command.

```
No configured and down cpus were selected.
```

Cause. No configured and nonrunning CPUs were specified in the RELOAD command.

Effect. No CPUs are reloaded.

Recovery. Reissue the command, including the configured and nonrunning CPU.

```
Operand error.
```

Cause. The operand was unacceptable.

Effect. The RELOAD command aborts.

Recovery. Correct and reissue the command.

```
Scanner error:nnn.
```

Cause. There was a scanner error scanning the command as indicated by the decimal number *nnn*.

Effect. The RELOAD command aborts.

Recovery. Correct or simplify the command and reissue it.

```
Syntax error:
(command line)
^
```

Cause. There was a syntax error in the indicated command line at or before the point shown by the arrow.

Effect. The command is ignored.

Recovery. Correct and reissue the command.

```
Warning: Alternate OSIMAGE SYSNN does not match reloader's
SYSNN.
```

Cause. The alternate system-image SYS *nn* does not match the SYS nn of the reloader.

Effect. The CPU is not reloaded.

Recovery. Specify a different CPU or none.

EMS Messages

These Event Management Service (EMS) messages can be generated by a TACL process.

001

```
TACL DEVICE I/O ERROR: file-system-error, device-name
```

file-system-error

the I/O error.

device-name

the device name.

Cause. An I/O error occurred when a TACL process attempted to communicate with a device. The message is generated only for the first occurrence of the error. For more information about file-system error messages, see the *Guardian Procedure Calls Reference Manual* and the *Guardian Procedure Errors and Messages Manual*.

This error is defined as ZTAC-EVT-IO-ERROR in the ZSPIDEF.ZTACDDL file.

Effect. Depends on the error.

Recovery. Depends on the error.

002

```
TACL BACKUP CREATE ERROR: process-create-error, error-detail
```

process-create-error

the process-creation error.

error-detail

the process-creation error detail.

Cause. A process creation error occurred when a TACL process attempted to create a backup process. The message is generated only for the first occurrence of the error. For more information about file-system error messages, see the *Guardian Procedure Calls Reference Manual* and the *Guardian Procedure Errors and Messages Manual*.

This error is defined as ZTAC-EVT-CREATE-ERROR in the ZSPIDEF.ZTACDDL file.

Effect. Depends on the error.

Recovery. Depends on the error.

Error Numbers

Some TACL functions (#ERRORNUMBERS in particular) return an error number as part of their result. Table B-1 on page B-50 lists the TACL error numbers and their meanings. More complete descriptions of the errors can be found earlier in this section.

Numbers lower than 1024 are issued by the file system or the sequential I/O facility (SIO) and are described elsewhere. If you receive an error greater than 1024 that is not included in this table, call your service provider.

Table B-2. Error Numbers Associated With TACL Messages (page 1 of 5)

Error Number	Error Text
1024	Missing close bracket
1025	Missing open bracket
1026	Text buffer overflow
1027	No routine has been called
1028	String variable may not contain more than one line
1029	Directory contents cannot be set
1030	Nondirectory may only be at end of variable name
1031	No such variable
1032	Nonexistent directory in variable name
1033	Badly formed variable name
1034	One or more errors occurred while loading library file
1035	File is neither a program nor a TACL macro
1036	Option may not appear more than once
1037	Arithmetic overflow
1038	Number or numeric variable expected
1039	Variable is not numeric
1040	Variable does not exist
1041	Disallowed by $CMON
1042	Variable does not have that level
1043	Duplicate keyword
1044	Unable to express a file name in network form
1045	Requires a numeric argument
1046	Delta syntax error
1047	Unable to expand macro

Table B-2. Error Numbers Associated With TACL Messages (page 2 of 5)

Error Number	Error Text
1048	Syntax error (any “Expecting ..." message)
1049	Too many options
1050	File error on previous out file
1051	Too many PARAMs
1053	Was not pushed
1054	Unable to express default volume in network form
1055	No level number or STRUCT qualification allowed
1056	Too long
1057	Multiply defined label in enclosure
1058	Badly formed label
1059	Unable to perform edit
1060	Unknown DELTA command
1061	Not found
1062	Not in buffer
1063	Unable to allocate contiguous word space
1064	Requires a specific level
1065	Invalid comment format
1066	Unable to load SECTION
1067	No such line
1068	Neither case label nor OTHERWISE found
1069	Enclosure not allowed in unbracketed line
1070	Level is in use
1071	All possible simultaneous servers and requesters in use
1073	System not available
1074	Invalid user name or password
1075	SETMYTERM operating system procedure failed
1076	Option conflicts with another option
1077	Unable to allocate space in segment
1078	Illegal level number syntax
1079	Variable is in a shared segment making it read-only
1080	Illegal variable name syntax
1081	Cannot push or pop the root segment's root
1082	BODY label not found
1083	This type of variable is inappropriate here

Table B-2. Error Numbers Associated With TACL Messages (page 3 of 5)

Error Number	Error Text
1084	Invalid alias format
1085	Too many successive aliases or alias loop
1086	Cannot resolve alias
1087	Not a legal variable type or I/O mode
1088	Too many arguments
1089	TACL process must be named
1090	Stack overflow
1091	Unknown function key
1092	Syntax error in lower bound
1093	Routine stack overflow
1094	Duplicate exception in this statement
1095	Too many simultaneous exceptions
1096	Syntax error in upper bound
1097	All block buffers in use
1098	Range must be last
1099	A non-STRUCT can appear only at the end
1101	Any #NEWPROCESS error (C-series only)
1107	Reference out of defined bounds
1108	CONVERTPROCESSTIME argument too large (>3.7 years)
1109	? line not allowed here
1110	Illegal internal character representation
1111	REDEFINE item would extend beyond item being redefined
1112	Item not found in STRUCT
1113	Segment file code must be 440
1114	Segment data structure invalid
1115	Segment version incompatible
1116	TACL not named, cannot have a backup
1117	Illegal CPU number
1118	Backup CPU may not be same as primary CPU
1119	Backup CPU is down
1120	No backup
1121	STRUCT's data would exceed 5000 bytes
1122	SPI buffer does not begin with -28
1123	STRUCT's data area is not long enough

Table B-2. Error Numbers Associated With TACL Messages (page 4 of 5)

Error Number	Error Text
1124	Token map would overflow its STRUCT
1125	Token map contains a specification unknown to TACL
1126	STRUCT is not long enough to be nulled by given token map
1127	Any DEFINE-oriented error
1128	Missing close quote; must be on same line as open quote
1129	You have no INLINE process
1130	SUPER.SUPER may not LOGON remotely
1131	Backup process already exists
1132	LIKE may not refer to any part of STRUCT containing it
1133	Segment is inconsistent because its last DETACHSEG failed to complete
1135	Item name already in use
1136	Unable to access userid file
1137	Illegal value
1138	You already have a current INLINE process
1139	You cannot explicitly set this variable
1140	Attribute value too long for TACL
1141	REDEFINE attempted to place word-aligned item on odd-byte boundary
1142	CPRULES file is corrupt
1143	Unsupported CPRULES file version
1144	Specified character class not present in CPRULES file
1145	CPRULES file size exceeds TACL buffer
1146	CPRULES file size exceeds CPRO buffer
1147	CPRULES filename error
1148	CPRULES file must be file code 199
1149	D-series newprocess error
1150	PROCESS_GETINFO_ error = n, error^detail = d
1151	PROCESS_GETINFOLIST_ error = n, error^detail = d
1152	PROCESS_GETPAIRINFO_ error = n, error^detail = d
1153	PROCESS_SETSTRINGINFO_ error = n, error^detail = d
1154	Expecting |WHILE| or |DO|
1155	Expecting |DO|
1156	Expecting |UNTIL|
1157	Enclosure not allowed in #DEF of DIRECTORY or STRUCT
1158	|OTHERWISE|, if present, must be the last label in #CASE

Table B-2. Error Numbers Associated With TACL Messages (page 5 of 5)

Error Number	Error Text
1159	TACL internal buffer too small
1160	This built-in variable cannot be pushed or popped
1161	Enclosure not allowed in this context
1162	Could not open window
1163	Expecting |THEN| or |ELSE|
1164	Label may not appear more than once
1165	Refer to #SETCONFIGURATION Built-In Function on page 9-347 for a complete description of this error. There is no text output for this error.
1166	Record size cannot exceed 1024 bytes for unstructured non-edit files
1167	Maximum number of attached segment files exceeded
1168	System's base address for selectable segments has been changed
2059	Current attribute set is incomplete and inconsistent
2060	No more DEFINEs
2061	No more attributes
2062	Illegal attribute name
2064	Required attribute-cannot be reset
2066	Required parameter is not supplied
2067	Illegal value for attribute
2068	Illegal class name
2069	DEFINE name not allowed under current DEFMODE setting
2073	Attempt to redefine =_DEFAULTS to another class
2074	DEFINE cannot be deleted
2049	Illegal DEFINE 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	DEFINE bounds error
2055	There is no attribute "attr" for the current class
2056	Attribute missing
2057	Current attribute set is incomplete
2058	Current attribute set is inconsistent, check number num

C

Mapping TACL Built-In Functions to Guardian Procedures

Many TACL built-in functions access Guardian procedures to provide functionality. Table C-1 lists the TACL built-in functions and specifies whether each TACL built-in function accesses a Guardian procedure, and if so, which procedure or procedures.

Table C-1. TACL Built-In Functions and Guardian Procedures (page 1 of 7)

TACL Built-In	Guardian Procedure
#ABEND	PROCESS_STOP_
#ABORTTRANSACTION	ABORTTRANSACTION
#ACTIVATEPROCESS	PROCESS_ACTIVATE_
#ADDDSTTRANSITION	ADDDSTTRANSITION
#ALTERPRIORITY	PROCESS_SETINFO_
#APPEND	None
#APPENDV	None
#ARGUMENT	None
#BACKUPCPU	PROCESS_GETINFO_, PROCESSHANDLE_DECOMPOSE_
#BEGINTRANSACTION	BEGINTRANSACTION
#BREAKPOINT	None
#BUILTINS	None
#CASE	None
#CHANGEUSER	PROCESSACCESSID
#CHARADDR	None
#CHARBREAK	None
#CHARCOUNT	None
#CHARDEL	None
#CHARFIND	None
#CHARFINDR	None
#CHARFINDRV	None
#CHARFINDV	None
#CHARGET	None
#CHARGETV	None
#CHARINS	None
#CHARINSV	None

Table C-1. TACL Built-In Functions and Guardian Procedures (page 2 of 7)

TACL Built-In	Guardian Procedure
#COLDLOADTACL	None
#COMPAREV	None
#COMPUTE	None
#COMPUTEJULIANDAYNO	COMPUTEJULIANDAYNO
#COMPUTETIMESTAMP	COMPUTETIMESTAMP
#COMPUTETRANSID	COMPUTETRANSID API
#CONTIME	The time is computed with conversion logic.
#CONVERTPHANDLE	PROCESSHANDLE_TO_STRING_
#CONVERTPROCESSTIME	CONVERTPROCESSTIME
#PROCESSINFO	FILENAME_DECOMPOSE_, PROCESSHANDLE_DECOMPOSE_, PROCESS_GETPAIRINFO_, PROCESS_GETINFOLIST_, PROCESS_GETINFO_, PROCESSHANDLE_TO_STRING_, FILENAME_EDIT_, FILENAME_DECOMPOSE_
#CONVERTTIMESTAMP	CONVERTTIMESTAMP
#CREATEFILE	FILE_CREATELIST_, FILENAME_DECOMPOSE_
#CREATEPROCESSNAME	CREATEPROCESSNAME
#CREATEREMOTENAME	CREATEREMOTENAME
#DEBUGPROCESS	PROCESS_DEBUG_
#DEF	None
#DEFINEADD	DEFINEADD
#DEFINEDELETE	DEFINEDELETE
#DEFINEDELETEALL	DEFINEDELETEALL
#DEFINEINFO	DEFINEINFO
#DEFINENAMES	GETSYSTEMNAME, DEFINENEXTNAME
#DEFINENEXTNAME	DEFINENEXTNAME
#DEFINEREADATTR	DEFINEREADATTR
#DEFINERESTORE	DEFINERESTORE
#DEFINERESTOREWORK	DEFINERESTOREWORK
#DEFINESAVE	DEFINESAVE
#DEFINESAVEWORK	DEFINESAVEWORK
#DEFINESETATTR	DEFINESETATTR
#DEFINESETLIKE	DEFINESETLIKE
#DEFINEVALIDATEWORK	DEFINEVALIDATEWORK
#DELAY	SIGNALTIMEOUT, AWAITIO

Table C-1. TACL Built-In Functions and Guardian Procedures (page 3 of 7)

TACL Built-In	Guardian Procedure
#DELTA	None
#DEVICEINFO	FILE_GETINFOLISTBYNAME_
#EMPTY	None
#EMPTYV	None
#EMSADDSUBJECT	EMSADDSUBJECT (EMS PROCEDURE)
#EMSADDSUBJECTV	EMSADDSUBJECT (EMS PROCEDURE)
#EMSGET	EMSGET (EMS PROCEDURE)
#EMSGETV	EMSGET (EMS PROCEDURE)
#EMSINIT	None
#EMSINITV	None
#EMSTEXT	EMSTEXT (EMS PROCEDURE)
#EMSTEXTV	EMSTEXT (EMS PROCEDURE)
#ENDTRANSACTION	ENDTRANSACTION API
#EOF	None
#ERRORTEXT	None
#EXCEPTION	None
#EXTRACT	None
#EXTRACTV	None
#FILEGETLOCKINFO	FILE_GETLOCKINFO_
#FILEINFO	FILEINQUIRE, FILENAME_UNRESOLVE_, FILEINFO, FILE_GETINFOLISTBYNAME_,
#FILENAMES	None
#FILTER	None
#FRAME	None
#GETCONFIGURATION	None
#GETPROCESSSTATE	PROCESS_GETINFOLIST_
#GETSCAN	None
#HISTORY	None
#IF	None
#INITTERM	None
#INLINEEOF	None
#INPUT	None
#INPUTV	None
#INTERACTIVE	FNAMECOMPARE
#INTERPRETJULIANDAYNO	INTERPRETJULIANDAYNO

Table C-1. TACL Built-In Functions and Guardian Procedures (page 4 of 7)

TACL Built-In	Guardian Procedure
#INTERPRETTIMESTAMP	INTERPRETTIMESTAMP
#INTERPRETTRANSID	INTERPRETTRANSID (TM/MP API)
#JULIANTIMESTAMP	JULIANTIMESTAMP
#KEEP	None
#KEYS	None
#LINEADDR	None
#LINEBREAK	None
#LINECOUNT	None
#LINEDEL	None
#LINEFIND	None
#LINEFINDR	None
#LINEFINDRV	None
#LINEFINDV	None
#LINEGET	None
#LINEGETV	None
#LINEINS	None
#LINEINSV	None
#LINEJOIN	None
#LOAD	None
#LOCKINFO	PROCESSHANDLE_TO_FILENAME_, FILENAME_TO_OLDFILENAME_, LOCKINFO
#LOGOFF	None
#LOOKUPPROCESS	PROCESSHANDLE_DECOMPOSE_, FILENAME_DECOMPOSE_, PROCESS_GETPAIRINFO_, GETSYSTEMNAME,
#LOOP	None
#MATCH	None
#MOM	PROCESS_GETINFO_
#MORE	None
#MYGMOM	PROCESS_GETINFO_
#MYPID	PROCESSHANDLE_DECOMPOSE_
#MYSYSTEM	MYSYSTEMNUMBER
#NEWPROCESS	FILENAME_DECOMPOSE_, PROCESS_GETINFO_, OLDFILENAME_TO_FILENAME_, PROCESS_CREATE_, PROCESSHANDLE_TO_STRING_, FILE_OPEN_, FILE_CLOSE_

Table C-1. TACL Built-In Functions and Guardian Procedures (page 5 of 7)

TACL Built-In	Guardian Procedure
#NEXTFILENAME	NEXTFILENAME, FNAMECOLLAPSE
#OPENINFO	FILENAME_DECOMPOSE_, FILE_GETOPENINFO_, FILENAME_UNRESOLVE_
#OUTPUT	None
#OUTPUTV	None
#PAUSE	PROCESSHANDLE_TO_STRING_, AWAITIO, FILEINFO
#POP	None
#PROCESS	MYSYSTEMNUMBER, PROCESSHANDLE_DECOMPOSE_, PROCESSHANDLE_TO_STRING_
#PROCESSEXISTS	None
#PROCESSINFO	PROCESS_GETPAIRINFO_, PROCESS_GETINFOLIST_ , PROCESS_GETINFO_ , PROCESSHANDLE_TO_STRING_ , FILENAME_DECOMPOSE_, FILENAME_EDIT_, PROCESSHANDLE_DECOMPOSE_
#PROCESSORSTATUS	PROCESSORSTATUS, REMOTEPROCESSORSTATUS
#PROCESSORTYPE	PROCESSOR_GETNAME_ , PROCESSHANDLE_DECOMPOSE_
#PURGE	PURGE, FILEINFO
#PUSH	None
#RAISE	None
#RENAME	FILE_GETINFOLISTBYNAME_, FILE_OPEN_, FILEINFO, FILE_RENAME_, CLOSE
#REPLY	None
#REPLYV	None
#REQUESTER	DEVICEINFO2, OLDFILENAME_TO_FILENAME_, FILENAME_COMPARE_, FILEINFO
#RESET	None
#REST	None
#RESULT	None
#RETURN	None
#ROUTINENAME	None
#SEGMENT	None
#SEGMENTCONVERT	None
#SEGMENTINFO	None
#SEGMENTVERSION	OPEN, FILEINFO, READ, CLOSE

Table C-1. TACL Built-In Functions and Guardian Procedures (page 6 of 7)

TACL Built-In	Guardian Procedure
#SERVER	FILENAME_TO_OLDFILENAME_
#SET	None
#SETBYTES	None
#SETCONFIGURATION	PROCESSACCESSID, FILE_GETINFOBYNAME_, FILE_OPEN_, FILE_CLOSE_, READX, POSITION, WRITEX
#SETMANY	None
#SETPROCESSSTATE	PROCESS_SETINFO_
#SETSCAN	None
#SETSYSTEMCLOCK	SETSYSTEMCLOCK
#SETV	None
#SHIFTSTRING	SHIFTSTRING
#SORT	No GPC Used
#SPIFORMATCLOSE	SPI_FORMAT_CLOSE_ (DSM PROCEDURE)
#SSGET	SSGET (EMS PROCEDURE)
#SSGETV	SSGET (EMS PROCEDURE)
#SSINIT	SSINIT (SPI PROCEDURE)
#SSMOVE	SSMOVE (SPI PROCEDURE)
#SSNULL	SSNULL (SPI PROCEDURE)
#SSPUT	SSPUT (SPI PROCEDURE)
#SSPUTV	SSPUT (SPI PROCEDURE)
#STOP	PROCESS_STOP_, PROCESS_GETPAIRINFO_
#SUSPENDPROCESS	PROCESS_SUSPEND_
#SWITCH	PROCESS_STOP_, CHECKPOINT
#SYSTEM	None
#SYSTEMNAME	GETSYSTEMNAME
#SYSTEMNUMBER	LOCATESYSTEM
#TACLOPERATION	None
#TACLVERSION	None
#TIMESTAMP	None
#TOSVERSION	TOSVERSION
#UNFRAME	None
#USERID	None
#USERNAME	None
#VARIABLEINFO	None

Table C-1. TACL Built-In Functions and Guardian Procedures (page 7 of 7)

TACL Built-In	Guardian Procedure
#VARIABLES	No GPC Used.
#VARIABLESV	No GPC Used.
#WAIT	No GPC Used.
#XFILEINFO	FILE_GETINFOLISTBYNAME_, OLDFILENAME_TO_FILENAME_ , FILENAME_DECOMPOSE_
#XFILENAMES	No GPC Used.
#XFILES	No GPC Used.
#XLOGON	PROCESSACCESSID, USER_AUTHENTICATE_
#XPPD	PROCESS_GETPAIRINFO_, MYSYSTEMNUMBER, PROCESSHANDLE_DECOMPOSE_, FILENAME_DECOMPOSE_
#XSTATUS	PROCESS_GETINFO_, PROCESS_GETPAIRINFO_, PROCESS_GETINFOLIST_, PROCESS_STOP_, UNUMZ, SHIFTSTRING, GETSYSTEMNAME, PROCESSHANDLE_DECOMPOSE_

Glossary

access mode. A file attribute that determines what operations you can perform on the file, like reading and writing.

alias. An alternative name for a given function.

ancestor. The process that is notified when a named process or process pair is deleted. The ancestor is usually the process that created the named process or process pair.

argument. A parameter that you specify when you invoke a macro or routine.

array data item. A portion of a STRUCT that is treated as an array; that is, you can refer to the whole item, or you can refer to individual elements of it.

ASSIGN. An association of a physical file name with a logical file name made by the TACL ASSIGN command. The physical file name is any valid file name. The logical file name is used within a program. The ASSIGN is therefore used to pass file names to programs.

Blade Element. See slice. Also known as a NonStop Blade Element.

BREAK mode. A mode of process execution where a process gains exclusive access to a terminal when the BREAK key is pressed. BREAK mode is established using SETPARAM function 3 or SETMODE function 11.

BREAK owner. The process that receives the break-on-device message when the BREAK key is pressed. The establishment of BREAK ownership is achieved using SETPARAM function 3 or SETMODE function 11.

breakpoint. A location (or point) in a program where execution is to be suspended so that you can then examine and perhaps modify the state of the program. You can set and clear breakpoints with _DEBUGGER commands.

built-in. A function or variable built into TACL; a built-in cannot be modified. Other variables can be modified by the user.

C-series system. A system that is running a C-series RVU.

CAID. See creator access ID (CAID).

child process. A process created by the current process.

code segment. An area of memory that contains program instructions to be executed, plus related information. An absolute segment whose logical pages are read from but never written back to the swap file. command. A text string that directs the computer to perform a task. Commands are usually composed of a verb that tells the computer what to do and an object or list of objects that is acted on by the verb. TACL commands are interpreted by TACL and are extensible.

command-interpreter monitor ($CMON). A server process that monitors requests made to the TACL process and affects the way TACL responds.

completion code. A value used to return information about a process to its ancestor process when the process is deleted. This value is returned in the process deletion message, system message -101.

condition code. A status returned by some file-system procedure calls to indicate whether the call was successful. A condition-code-greater-than (CCG) indicates a warning, a condition-code-less-than (CCL) indicates an error, and a condition-code-equal (=) indicates successful execution.

conversational mode. A mode of communication between a terminal and its I/O process in which each byte is transferred from the terminal to the processor I/O buffer as it is typed. Each file-transfer operation finishes when a line-termination character is typed at the terminal. Contrast with page mode.

creator. The process that initiates execution of another process. Compare with mom and ancestor.

creator access ID (CAID). A process attribute that identifies, by user ID, the user who initiated the process creation. Contrast with process access ID.

data segment. A type of absolute segment whose logical pages contain information to be processed by the instructions in the related code segment.

deadlock. A situation in which two processes or two transactions cannot continue because they are each waiting for the other to release a lock.

default process. The process whose name is returned by the #PROCESS function. It is the process most recently created by a RUN or RUND command, an implied RUN, or a #NEWPROCESS built-in function, or for which TACL was most recently paused by a PAUSE proc-spec command or a #PAUSE proc-spec built-in function; if that process is no longer running, there is no default process.

DEFINE. A named set of attributes and values.

DEFINE name. An identifier preceded by an equal sign that can be used in place of an actual name to identify a DEFINE in a procedure call. See DEFINE.

Delta. The low-level character editor provided by TACL.

destination control table (DCT). A collection of operating system data structures that serves as a directory of named processes and logical devices.

device. A peripheral hardware attachment used for input and output; for example, a printer or a disk.

device subtype. A value that further qualifies a device type. For example, a device type of 4 indicates a magnetic tape drive; if the same device has a device subtype of 2, then the magnetic tape drive has a 3206 controller.

disk volume. Also called a disk or a volume; a magnetic storage medium. Disk names consist of a dollar sign ($) followed by one to seven alphanumeric characters (network) or one to eight alphanumeric characters (local), the first of which must be alphabetic.

EDIT file. . A file in a format defined by the EDIT product.

enclosure. A unit composed of one or more labels, such as |THEN| or |DO|, and the text associated with each label. Enclosures are found only in the TACL built-in functions #DEF, #IF, #CASE, and #LOOP, which are enclosed in brackets to provide boundaries for their enclosures. TACL defers execution of text that is associated with labels until it determines the correct label to use.

Enscribe. A database record management system.

exception. An unusual event that causes TACL to interrupt the normal flow of invocations and transfer to special code. (See exception handler.) This unusual event could be BREAK, a TACL error, or a user-defined exception.

exception handler. A series of TACL statements that perform resource deallocation and cleanup after an exception.

exclusion mode. The attribute of a lock that determines whether any process except the lock holder can access the locked data.

expand. A type of invocation. (See invoke.) To expand a variable, specify the variable name in brackets; TACL returns the expansion in place of the variable name.

expression. A text, string, or integer constant, a variable, or a value obtained by combining constants, variables, and other expressions with operators. Expressions are used as arguments to commands and built-in functions.

extended data segment. One or more consecutive absolute segments that are dynamically allocated by a process.

extensible data segment. An extended data segment for which swap file extents are not allocated until needed.

extent. A contiguous area of a disk allocated to the same file.

fault tolerance. The ability of a computer system to continue operating during and after a fault (the failure of a system component).

file code. An integer value assigned to a file for application-dependent purposes.

file lock. A mechanism that restricts access to a file by all processes except the lock owner.

file. As used here, a file refers to an organized collection of data stored on a disk. In general, a file can be a disk file, a process, or a device.

file name. A unique name for a file. This name is used to open a file and thereby provides a connection between the opening process and the file. File names consist of one to eight alphanumeric characters, the first of which must be alphabetic. file name template. A sequence of characters including the asterisk (*) and question mark (?) that matches existing file names by expanding each asterisk to zero or more letters, digits, dollar signs ($), and pound signs (#) and replacing each question mark with exactly one letter, digit, dollar sign, or pound sign.

file system. A set of operating system procedures and data structures that provides for communication between a process and a file, which can be a disk file, a device other than a disk, or another process.

FILLER byte. A portion of a STRUCT that is used only to maintain the alignment of adjacent STRUCT items.

frame. A local environment managed by the #FRAME, #UNFRAME, and #RESET built-in functions.

fully qualified file name. The complete name of a file, including the node name. For permanent disk files, this file name consists of a node name, volume name, subvolume name, and file ID. For temporary disk files, the file name consists of a node name, a subvolume name, and a temporary file ID. For a device, the file name consists of a node name and a device name or logical device number. For a named process, the file name consists of a node name, and a process name. For an unnamed process, the file name consists of a node name, CPU number, PIN, and sequence number. Contrast with partially qualified file name.

function. An operation or set of operations that is invoked by the appearance of the function name and its arguments at the point where the result of the function is wanted. A built-in function is hard coded into TACL; users can define other functions. Variable types for functions include TEXT, MACRO, and ROUTINE.

GMT. See Greenwich mean time (GMT).

godmother. See job ancestor.

Greenwich mean time (GMT). The mean solar time for the meridian at Greenwich, England.

Gregorian date. A date specified according to the common calendar using the month of the year (January through December), the day of the month, and the year A.D.

home terminal. The terminal whose name is returned by a call to the MYTERM procedure, or the name returned in the hometerm parameter of the PROCESS_GETINFO_ procedure. The home terminal is often the terminal from which the process or process's ancestor was started.

interprocess communication (IPC). The exchange of messages between processes in a system or network. interrupt. The mechanism by which a processor module is notified of an asynchronous event that requires immediate processing.

invoke. A request to execute TACL code. To invoke a variable, (1) list its name (like an implied RUN statement) without regard to results or (2) surround the variable name in square brackets ([]) to replace the name with its expansion (text or macro variable) or results (routine).

IPC. See interprocess communication (IPC).

job ancestor. A process that is notified when a process that is part of a job is deleted. The job ancestor of a process is the process that created the job to which the process belongs.

Julian timestamp. The number of microseconds since midnight January 1, 4713 B.C. at the Greenwich meridian.

LCT. See local civil time (LCT).

LDEV. See logical device (LDEV).

level. One element of the set of values stored in a stack and known as a variable.

local civil time (LCT). Wall-clock time in the current time zone, including any compensation for daylight-saving time.

local standard time (LST). The time of day in the local time zone excluding any compensation made for daylight-saving time.

logical device (LDEV). (1) An addressable device, independent of its physical environment. Portions of the same logical device can be located in different physical devices, or several logical devices or parts of logical devices can be located in one physical device. (2) A process that can be accessed as if it were an I/O device; for example, the operator process is logical device LDEVOPR.

logical device number. A number that identifies a configured logical device. A logical device number can be used instead of a device file name when opening a device file.

logical processor. The combination of equivalent processor elements in the processor slices that are running in the same instruction stream in loose lock-step.

LST. See local standard time (LST).

macro. A named sequence of one or more instructions invoked by the appearance of the macro name. When a macro is invoked, TACL replaces arguments of the form %n% with actual arguments passed to it and returns, as a result, the instructions that define the macro, including argument values.

message system. A set of operating system procedures and data structures that handles the mechanics of exchanging messages between processes.

metacharacter. A character that directs TACL to evaluate subsequent text in a special way.

mom. A process that is notified when certain other processes are deleted. When a process is part of a process pair, the mom of the process is the other member of the pair. When a process is unnamed, its mom is usually the process that created it. monitor. A process that, among other functions, is responsible for checking that certain other processes continue to run. If a process should stop, it is the monitor's responsibility to restart it.

multibyte character set. A means for identifying written characters for national languages that require more than one byte to represent a single character.

named process. A process to which a process name was assigned when the process was created. Contrast with unnamed process.

node. A system of one or more processor modules. Typically, a node is linked with other nodes to form a network.

node name. The portion of a file name that identifies the system through which the file can be accessed.

nonretryable error. An error condition returned by the file system that cannot be recovered by retrying the operation even after operator intervention. Contrast with retryable error.

NonStop Advanced Architecture. The H-series architecture that allows up to three levels of redundancy. Logical processors consist of one to three physical processors known as processor elements (PEs). The logical processor is equivalent to the term CPU. For availability issues, the processor elements are all located on different circuit boards. These boards are known as blade elements (or slices) and are identified by the letters A, B, or C.

NonStop Blade Element. See slice. Also known as Blade Element.

NonStop SQL/MP. A relational database management system that provides efficient online access to large distributed databases.

nowait I/O. An operation with an I/O device where the process does not wait for the I/O operation to finish. Contrast with waited I/O.

NSAA. See NonStop Advanced Architecture.

one-way communication. A form of interprocess communication where the sender of a message (the requester) does not expect any data in the reply from the receiver of the message (the server). Contrast with two-way communication.

operator. Perform mathematical or logical operations on values.

page. 1,024 words of contiguous data.

page mode. A mode of communication between a terminal and its I/O process in which the terminal stores up to a full page of data (1,920 bytes) in its own memory before sending the page to the I/O process. Contrast with conversational mode.

PAID. See process access ID (PAID).

parallel (method of dumping). The NSAA design of multiple physical processors linked together as a logical processor allows for parallel operations. A single processor element, omitted from a reload of a logical processor, can be dumped and reintegrated without affecting the other PEs in that logical processor. In this way the NonStop operating system can perform a dump and continue application-level processing at the same time.

PARAM. An association of an ASCII value with a parameter name made by the TACL

PARAM command. You can use PARAMs to pass parameter values to processes. partially qualified file name. A file name in which only the right-hand file name parts are specified. The remaining parts of the file name assume default values. Contrast with fully qualified file name.

partitioned file. A logical file made up of several partitions that can reside on different disks. Generic key values determine the partition on which a given record resides. permanent disk file. A file that remains on disk until it is explicitly purged.

PE. See processor element

PFS. See process file segment (PFS).

PIN. See process identification number (PIN).

primary extent. The first contiguous area of disk allocated to a file. See also secondary extent.

priority. An indication of the precedence with which a process gains access to the instruction processing unit.

process. A program that has been submitted to the operating system for execution.

processor element. A single microprocessor or microprocessor core, with its associated memory, capable of executing a single instruction stream.

process access ID (PAID). A user ID used to determine whether a process can make requests to the system; for example, to open a file, stop another process, and so on. The process access ID is usually the same as the creator access ID, but it can be different; the owner of the corresponding object file can set the object file security such that it runs with a process access ID equal to the user ID of the file owner, rather than the creator of the process. Contrast with creator access ID.

process file name. A file name that identifies a process.

process file segment (PFS). An extended data segment that is automatically allocated to every process and contains operating system data structures, file-system data structures, and memory-management pool data structures.

process ID. A C-series structure that serves as an address of a process.

process identification number (PIN). An unsigned integer that identifies a process in a processor module.

process name. A name that can be assigned to a process when the process is created. A process name uniquely identifies a process or process pair in a system. A process name consists of a dollar sign ($), followed by one to five alphanumeric characters, the first of which must be alphabetic.

process pair. Two processes created from the same object file running in a way that makes one process a backup process of the other in case of failure. Periodic checkpointing ensures that the backup process is always ready to take over from the primary if the primary process should fail. The process pair has one process name, but each process has a different process identification number (PIN).

process qualifier. A suffix to a process file name that gets passed to a process when the process is opened; its use is application-dependent.

process time. The amount of time a process has been active while the processor module was in the environment of the process.

processor clock. A hardware timer on each processor module that keeps processor time; the number of microseconds since cold load.

processor time. The time represented by a processor clock.

program. A sequence of instructions and data. In TACL, variables of type TEXT, MACRO, and ROUTINE can define programs.

real time. See wall-clock time.

record lock. A lock held by a process or a transaction that restricts access to that record by other processes.

redefinition. A STRUCT declaration that gives a new definition, such as a different data type or a different alignment, to an existing STRUCT or STRUCT item. All definitions are valid concurrently, allowing a STRUCT or STRUCT item to be used in a variety of ways.

reply. A response to a requester process by a server process. Contrast with request.

request. A message formatted and sent to a server by a requester. Requests also include status messages such as CPU up and CPU down messages, which are placed on the intended recipient's process message queue ($RECEIVE file) by the operating system. Contrast with reply.

requester. A process that initiates interprocess communication by sending a request to another process. Contrast with server.

response. See reply.

retryable error. An error condition returned by the file system that can be corrected by repeating the operation that caused the error. Sometimes operator intervention is required before the retry; for example, to put paper into an empty printer. Contrast with nonretryable error.

secondary extent. A contiguous area of disk storage allocated to a file. A file is made up of one or more extents; the first extent is the primary extent, and other extents are secondary extents. The secondary extents are all the same size for a specific file; the primary extent can be a different size. See also primary extent.

segment. A unit of storage consisting of up to 64 pages of 1,024 words each.

segment file. As used by TACL, a file accessible by TACL that can contain TACL code and data.

server. The process that receives, acts upon, and replies to messages from requesters. Contrast with requester.

shared data segment. An extended data segment that can be accessed by more than one process.

simple data item. A STRUCT item that contains a single value of a specific type.

slice. A portion of one or more logical processors and one part of a processor complex. A slice consists of a chassis, processor board containing one or more processor elements and memory, I/O interface board, midplane, optics adapters, fans, and power supplies. Also called a *processor slice*. Also known as a *blade element.*

space-separated list. A list whose entries are separated from each other by a space. Several built-in functions accept space-separated lists of values.

startup sequence. A convention for sending and receiving certain messages while starting a new process. By convention, the new process receives an Open message, followed by a startup message, an assign message for each ASSIGN in effect, a param message if there are any PARAMs in effect, and then a Close message string. A type of argument that some commands and functions accept in place of a variable. A string can be the name of a variable, text enclosed in quotation marks, or a concatenation of such entities. The concatenation operator is '+' (the single quotes are part of the operator). Under control of the QUOTED input format, a quoted string can contain TACL metacharacters.

STRUCT. A variable that is structured into individual components that can be accessed individually. Items within a STRUCT can be simple data items, arrays (which can be further broken down into individual elements), or substructures.

STRUCT item. An element of a structure that can be individually accessed by a name of the form structure-name:item-name.

substructure. A STRUCT item that is itself a STRUCT.

subvolume. A group of files stored on disk. These files all have the same subvolume name, but each has a different file ID. A subvolume name consist of one to eight alphanumeric characters, the first of which must be alphabetic. An example of a subvolume name is $DATA.INFO. An example of a file name in this subvolume is $DATA.INFO.RESULTS.

swapping. The process of copying information between physical memory and disk storage.

system process. A process whose primary purpose is to manage system resources rather than to solve a user’s problem. A system process is essential to a system-provided service. Failure of a system process often causes the processor module to fail. Most system processes are automatically created when the processor module is cold loaded. Contrast with user process.

system time. The time represented by any synchronized processor clock in the system.

template. A string of characters, including the special characters * and ?, used to match another string of characters. Templates can be used in place of file names and DEFINE names in some commands and built-in functions.

temporary disk file. A file stored on disk that will be purged automatically as soon as the process that created it stops.

terminal-simulation process. A process that is made to behave like a terminal file.

text. A set of characters from the ISO 8859.1 character set. The length of text can be limited by a specific function or command. TACL interprets a text argument as all remaining text on the line, with leading and trailing spaces and end-of-line characters removed.

timekeeping. A function performed by the operating system that involves initializing and maintaining the correct time in a processor module.

timestamp. An item containing a representation of the time. A timestamp can be applied to an object at a critical point, such as the last modification time of a file. transaction identifier. A four-word identifier that uniquely identifies a transaction within the Transaction Management Facility (TMF) subsystem.

TMF. See Transaction Management Facility (TMF).

Transaction Management Facility (TMF). HP software that provides transaction protection and database consistency in demanding online transaction processing (OLTP) and decision-support environments. It gives full protection to transactions that access distributed SQL and Enscribe databases, as well as recovery capabilities for transactions, online disk volumes, and entire databases.

transfer mode. The protocol by which data is transferred between a terminal and the computer system. See conversational mode and page mode.

two-way communication. A form of interprocess communication in which the sender of a message (requester process) expects data in the reply from the receiver (server process). Contrast with one-way communication.

unnamed process. A process to which a process name was not assigned when the process was created. Contrast with named process.

user ID. A unique pair of numbers that identify a user. A user ID has the form group-id,user-id, where the group-id identifies the user's group, and user-id identifies the user within the group.

user process. A process whose primary purpose is to solve a user's problem. A user process is not essential to the availability of a processor module and is created only when the user explicitly creates it. Contrast with system process.

variable. A named quantity that can assume any of a given set of values.

variable level. A portion of a variable that can be individually addressed. New levels can be added to the top of a variable stack, pushing down earlier levels, and can be popped off the top of the stack. When the last level is popped, the variable ceases to exist. For simplicity, variable levels are referred to as variables in many descriptions in this manual.

variable line. A portion of a variable level that ends with a binary zero (an internal end-of-line character). Lines can be removed from the beginning of a variable level with the #EXTRACT and #EXTRACTV functions and can be added to the end of a variable level with the #APPEND and #APPENDV function.

variable type. The designation (MACRO, DELTA, STRUCT, TEXT, and so on) of a variable level that describes its contents and the use for which it is designated.

virtual memory. A range of addresses that processes use to reference real storage, where real storage consists of physical memory and disk storage.

volume. A disk drive or a pair of disk drives that forms a mirrored disk.

waited I/O. An operation with an I/O device where the process waits until the operation finishes. Contrast with nowait I/O.

waiting process. A process that cannot execute until an event occurs, a resource becomes available, or an interval of time passes.

wall-clock time. The local time of day, including any adjustment for daylight-saving time.

working set. A collection of DEFINE attributes that have been assigned values.

$CMON. See command-interpreter monitor ($CMON).

$RECEIVE. A special file name through which a process receives messages from other processes.

Index

A

B

C

D

E

F

H

I

J

K

L

M

O

P

Q

R

S

T

U

V

W

X

Y

Special Characters