

# Common Run-Time Environment (CRE) Programmer's Guide

## Abstract

This manual describes the Common Run-Time Environment (CRE), a set of run-time services that enable system and application programmers to write mixed-language programs. In the TNS environment, the TNS CRE run-time services support C, COBOL, FORTRAN, and TAL language programs. In the native environment, the native CRE run-time services support C, C++, COBOL, and pTAL language programs.

## Product Version

CRE G09, CRE H01

## Supported Release Version Updates (RVUs)

This manual supports G06.25 and all subsequent G-series RVUs and H06.03 and all subsequent H-series RVUs until otherwise indicated by its replacement publication.

Part Number	Published
528146-004	February 2006

## Document History

Part Number	Product Version	Published
528146-003	CRE G09, CRE H01	July 2005
528146-004	CRE G09, CRE H01	February 2006

# Common Run-Time Environment (CRE) Programmer's Guide

**Glossary**

**Index**

**Examples**

**Figures**

**Tables**

<a href="#">What's New in This Guide</a>	xiii
<a href="#">Guide Information</a>	xiii
<a href="#">New and Changed Information</a>	xiii
<a href="#">About This Guide</a>	xv
<a href="#">Audience</a>	xvi
<a href="#">Organization</a>	xvi
<a href="#">Additional Information</a>	xvii
<a href="#">Hypertext Links</a>	xix

## 1. Introducing the CRE

<a href="#">Mixed-Language Programming Without the CRE</a>	1-1
<a href="#">What Is the CRE?</a>	1-2
<a href="#">Selecting a Run-Time Environment</a>	1-5
<a href="#">Advantages of Using the CRE</a>	1-8

## 2. CRE Services

<a href="#">Comparing the CRE in the OSS and Guardian Environments</a>	2-2
<a href="#">Standard Files</a>	2-2
<a href="#">\$RECEIVE</a>	2-3
<a href="#">Memory Organization</a>	2-3
<a href="#">Traps and Exceptions</a>	2-3
<a href="#">Program Initialization</a>	2-4
<a href="#">Program Termination</a>	2-4
<a href="#">Error Reporting</a>	2-5
<a href="#">Standard Functions</a>	2-5
<a href="#">CRE Services</a>	2-5
<a href="#">Process Pairs</a>	2-6
<a href="#">Writing TAL Routines That Use the TNS CRE</a>	2-6
<a href="#">Writing pTAL Routines That Use the Native CRE</a>	2-8
<a href="#">Program Initialization</a>	2-9
<a href="#">Designating a Main Routine</a>	2-9
<a href="#">TNS CRE Initialization</a>	2-10

## **2. CRE Services (continued)**

### Program Initialization (continued)

Native CRE Initialization 2-13

Initialization Errors 2-14

Initializing the TNS CRE From TAL 2-15

Initializing the Native CRE From pTAL 2-15

### Program Termination 2-15

CRE\_Terminator\_Procedure 2-16

Handling Error Conditions in CRE\_Terminator 2-16

### Sharing Standard Files 2-17

Sharing Standard Files Without Using the CRE 2-17

Sharing Standard Files Using the CRE 2-18

Using CRE Functions to Access the Standard Files 2-27

Determining When Standard Files Are Opened 2-28

Using Terminals and Process 2-28

Program Startup Message 2-28

Standard Input 2-29

Standard Output 2-31

Standard Log 2-33

### Using \$RECEIVE 2-34

\$RECEIVE and Program Initialization 2-35

Messages Received From \$RECEIVE 2-35

\$RECEIVE and the Languages Supported by the CRE 2-35

### Using a Spooler Collector 2-36

### Memory Organization 2-37

TNS CRE Memory 2-37

Native CRE Memory 2-41

### Using the Native Heap Managers 2-43

Undetected Logic Errors Can Exist in Code that Uses the Original Heap Manager 2-44

Using the Overwrite Feature to Detect Logic Errors 2-44

Using the Programmatic Heap-Management Attributes 2-45

### TNS CRE Traps and Exceptions 2-47

Errors in Program Logic 2-48

Hardware Traps 2-49

Catastrophic Errors 2-49

TNS CRE Trap Handler 2-50

Using ARMTRAP 2-51

### Writing Messages to Standard Log 2-51

## **2. CRE Services (continued)**

<a href="#">Language-Specific Error Handling</a>	2-52
<a href="#">C Routines</a>	2-52
<a href="#">COBOL Routines</a>	2-53
<a href="#">FORTRAN Routines</a>	2-54
<a href="#">TAL Routines</a>	2-54
<a href="#">pTAL Routines</a>	2-54
<a href="#">Reporting CRE Errors in the OSS Environment</a>	2-54
<a href="#">Native CRE Signals and Exceptions</a>	2-55
<a href="#">Using CRE Services</a>	2-56
<a href="#">Using Standard Functions</a>	2-56
<a href="#">CRE and RTL Prefixes</a>	2-57
<a href="#">Type Suffixes</a>	2-59
<a href="#">Using Process Pairs</a>	2-59
<a href="#">Requirements for Using Process Pairs</a>	2-59
<a href="#">Language Support for Process Pairs</a>	2-60
<a href="#">Using C Routines in Process Pairs</a>	2-60
<a href="#">Results of Operations That Support Process Pairs</a>	2-61
<a href="#">Using the Inspect, Native Inspect, and Visual Inspect Symbolic Debuggers With CRE Programs</a>	2-62
<a href="#">Selecting a Debugger</a>	2-63
<a href="#">Locating the Corrupter of TNS CRE Pointers</a>	2-64
<a href="#">Circumventing the CRE</a>	2-66

## **3. Compiling and Binding Programs for the TNS CRE**

<a href="#">Compiling Programs for the CRE</a>	3-1
<a href="#">Specifying a Run-Time Environment</a>	3-1
<a href="#">Sourcing-in CRELIB Function Declarations</a>	3-4
<a href="#">CRE Data Blocks</a>	3-5
<a href="#">Binding Programs for the CRE</a>	3-5
<a href="#">Run-Time Libraries</a>	3-7
<a href="#">Sample Binder Sessions</a>	3-8
<a href="#">Bind-Time Validation for Mixed-Language Programs</a>	3-8

## **4. Compiling and Linking Programs for the Native CRE**

<a href="#">Using the Environment Variable for C and C++ Modules</a>	4-1
<a href="#">Sourcing In CRE External Declarations for pTAL Modules</a>	4-2
<a href="#">Linking Modules</a>	4-3

## **5. Using the Common Language Utility (CLU) Library**

- [What Is the CLU Library?](#) 5-1
- [Compiling and Binding or Linking Programs That Use the CLU Library](#) 5-2
- [Creating Processes](#) 5-2
- [Locating and Identifying File Connectors](#) 5-2
- [Using the Saved Message Utility Functions](#) 5-2
  - [Services Provided by the Saved Message Utility](#) 5-4
  - [Content of Messages](#) 5-6
  - [Using SMU Routines to Manipulate Messages](#) 5-7
  - [Using the environ Array](#) 5-9

## **6. CRE Service Functions**

- [Environment Functions](#) 6-1
  - [CRE\\_Getenv](#) 6-1
  - [CRE\\_Putenv](#) 6-2
- [File-Sharing Functions](#) 6-4
  - [CRE\\_File\\_Close](#) 6-5
  - [CRE\\_File\\_Control](#) 6-6
  - [CRE\\_File\\_Input](#) 6-8
  - [CRE\\_File\\_Message](#) 6-9
  - [CRE\\_File\\_Open](#) 6-11
  - [CRE\\_File\\_Output](#) 6-20
  - [CRE\\_File\\_Retrycheck](#) 6-22
  - [CRE\\_File\\_Setmode](#) 6-23
  - [CRE\\_Hometerm\\_Open](#) 6-24
  - [CRE\\_Log\\_Message](#) 6-25
  - [CRE\\_Spool\\_Start](#) 6-27
- [\\$RECEIVE Functions](#) 6-30
  - [CRE\\_Receive\\_Open\\_Close](#) 6-31
  - [CRE\\_Receive\\_Read](#) 6-38
  - [CRE\\_Receive\\_Write](#) 6-41
- [CRE\\_Terminator](#) 6-42
- [Exception-Handling Functions](#) 6-44
  - [CRE\\_Log\\_GetPrefix](#) 6-45
  - [CRE\\_Stacktrace](#) 6-45

## **7. Math Functions**

- [Arithmetic Overflow Handling](#) 7-1
- [Standard Math Functions](#) 7-1

## **7. Math Functions (continued)**

<a href="#">Arccos</a>	7-4
<a href="#">Arcsin</a>	7-5
<a href="#">Arctan</a>	7-5
<a href="#">Arctan2</a>	7-6
<a href="#">Cos</a>	7-7
<a href="#">Cosh</a>	7-7
<a href="#">Exp</a>	7-8
<a href="#">Ln</a>	7-8
<a href="#">Log10</a>	7-9
<a href="#">Lower</a>	7-10
<a href="#">Mod</a>	7-11
<a href="#">Normalize</a>	7-12
<a href="#">Odd</a>	7-13
<a href="#">Positive_Diff</a>	7-13
<a href="#">Power</a>	7-15
<a href="#">Power2</a>	7-16
<a href="#">Random_Set, Random_Next</a>	7-17
<a href="#">Round</a>	7-17
<a href="#">Sign</a>	7-18
<a href="#">Sin</a>	7-19
<a href="#">Sinh</a>	7-19
<a href="#">Split</a>	7-20
<a href="#">Sqrt</a>	7-20
<a href="#">Tan</a>	7-21
<a href="#">Tanh</a>	7-21
<a href="#">Truncate</a>	7-22
<a href="#">Example</a>	7-22
<a href="#">Upper</a>	7-22
<a href="#">Sixty-Four-Bit Logical Operations (Bit Manipulation Functions)</a>	7-23
<a href="#">Return Value</a>	7-24
<a href="#">Examples</a>	7-24
<a href="#">Remainder</a>	7-25
<a href="#">Return Value</a>	7-25
<a href="#">Decimal Conversion Functions</a>	7-25
<a href="#">Decimal_to_Int</a>	7-26
<a href="#">Int_to_Decimal</a>	7-27

## **8. String and Memory Block Functions**

### String Functions 8-1

Atoi, Atol, Atof 8-3

Stcarg 8-4

Stccpy 8-5

Stcd\_I 8-6

Stcd\_L 8-7

Stch\_I 8-8

Stci\_D 8-9

Stcpm 8-10

Stcpma 8-11

Stcu\_D 8-12

Stpblk 8-13

Stpsym 8-14

Stptok 8-15

Strcat 8-16

Strchr 8-17

Strcmp 8-18

Strcpy 8-19

Strcspn 8-20

Strlen 8-21

Strncat 8-22

Strncmp 8-23

Strncpy 8-24

Strpbrk 8-25

Strrchr 8-26

Strspn 8-27

Strstr 8-27

Strtod 8-28

Strtol 8-29

Strtoul 8-30

Substring\_Search 8-32

### Memory Block Functions 8-33

Memory\_Compare 8-34

Memory\_Copy 8-35

Memory\_Findchar 8-36

Memory\_Move 8-37

Memory\_Repeat 8-38

Memory\_Set 8-39



## **8. String and Memory Block Functions (continued)**

Memory Block Functions (continued)

Memory\_Swap 8-40

## **9. Common Language Utility (CLU) Library Functions**

CLU\_Process\_Create 9-1

Return Value 9-4

COBOL Considerations 9-4

FORTRAN Considerations 9-8

CLU\_Process\_File\_Name 9-12

Return Value 9-14

COBOL Considerations 9-15

FORTRAN Considerations 9-16

SMU Functions 9-17

SMU\_Assign\_CheckName 9-18

SMU\_Assign\_Delete 9-19

SMU\_Assign\_GetText 9-21

SMU\_Assign\_GetValue 9-22

SMU\_Assign\_PutText 9-23

SMU\_Assign\_PutValue 9-25

SMU\_Message\_CheckNumber 9-26

SMU\_Param\_Delete 9-27

SMU\_Param\_GetText 9-28

SMU\_Param\_PutText 9-29

SMU\_Startup\_Delete 9-30

SMU\_Startup\_GetText 9-31

SMU\_Startup\_PutText 9-33

SMU Function Considerations 9-34

COBOL Considerations 9-35

FORTRAN Considerations 9-37

TAL Considerations 9-39

EpTAL Considerations 9-39

pTAL Considerations 9-40

## **10. Run-Time Diagnostic Messages**

Error Effects and Recovery 10-1

Format of Messages in This Section 10-2

Trap and Signal Messages 10-3

CRE Service Function Messages 10-6

## **10. Run-Time Diagnostic Messages (continued)**

<a href="#">Heap-Management Messages</a>	10-12
<a href="#">Function Parameter Message</a>	10-14
<a href="#">Math Function Messages</a>	10-15
<a href="#">Function Parameter Messages</a>	10-17
<a href="#">Input/Output Messages</a>	10-18
<a href="#">COBOL Messages</a>	10-25
<a href="#">FORTRAN Messages</a>	10-25
<a href="#">Native CRE Messages</a>	10-25
<a href="#">Mapping Message Numbers Between Run-Time Environments</a>	10-26

## **A. Data Type Correspondence**

### **Glossary**

### **Index**

## **Examples**

<a href="#">Example 2-1.</a>	<a href="#">C Program That Overwrites the MCB Pointer</a>	2-65
<a href="#">Example 2-2.</a>	<a href="#">Run of C Program That Overwrites the MCB Pointer</a>	2-65
<a href="#">Example 2-3.</a>	<a href="#">Inspect Session for C Program That Overwrites the MCB Pointer</a>	2-66

## **Figures**

<a href="#">Figure 1-1.</a>	<a href="#">Language-Specific Run-Time Environments</a>	1-1
<a href="#">Figure 1-2.</a>	<a href="#">The Common Run-Time Environment in the Guardian Environment</a>	1-2
<a href="#">Figure 1-3.</a>	<a href="#">The Common Run-Time Environment in the OSS Environment</a>	1-3
<a href="#">Figure 2-1.</a>	<a href="#">A C-Series Mixed-Language Process</a>	2-18
<a href="#">Figure 2-2.</a>	<a href="#">Using the CRE—Mixed-Language Process—Quiescent State</a>	2-20
<a href="#">Figure 2-3.</a>	<a href="#">Using the CRE—The COBOL Routine Defaults Opening Standard Output</a>	2-21
<a href="#">Figure 2-4.</a>	<a href="#">Using the CRE—The TAL Routine Opens Standard Output</a>	2-22
<a href="#">Figure 2-5.</a>	<a href="#">Using the CRE—The C Routine Opens Standard Output</a>	2-23
<a href="#">Figure 2-6.</a>	<a href="#">Using the CRE—Quiescent State With Standard Output Open</a>	2-24
<a href="#">Figure 2-7.</a>	<a href="#">Using the CRE—The COBOL Routine Writes to the File \$VOL.SUBVOL.FILE</a>	2-25
<a href="#">Figure 2-8.</a>	<a href="#">Using the CRE—The TAL Routine Writes to the File \$VOL.SUBVOL.FILE</a>	2-26
<a href="#">Figure 2-9.</a>	<a href="#">Using the CRE—The C Routine Writes to the File \$VOL.SUBVOL.FILE</a>	2-27

## Figures (continued)

<a href="#">Figure 2-10.</a>	<a href="#">Process Startup Message Layout</a>	2-29
<a href="#">Figure 2-11.</a>	<a href="#">Establishing the File Name of Standard Input</a>	2-30
<a href="#">Figure 2-12.</a>	<a href="#">Establishing the File Name of Standard Output</a>	2-32
<a href="#">Figure 2-13.</a>	<a href="#">Establishing the File Name of Standard Log</a>	2-34
<a href="#">Figure 2-14.</a>	<a href="#">Organization of a Small-Memory-Model Program Running in the TNS CRE</a>	2-38
<a href="#">Figure 2-15.</a>	<a href="#">Organization of a Large-Memory- or Wide-Memory Model Program Running in the TNS CRE</a>	2-39
<a href="#">Figure 2-16.</a>	<a href="#">Writing a CRE Error Message in the OSS Environment</a>	2-55
<a href="#">Figure 5-1.</a>	<a href="#">Messages Manipulated by the SMU</a>	5-5
<a href="#">Figure 6-1.</a>	<a href="#">Using Connections to Share a File Open</a>	6-16
<a href="#">Figure 6-2.</a>	<a href="#">Determining Spooler Buffering in CRE_File_Open</a>	6-20
<a href="#">Figure 6-3.</a>	<a href="#">Determining Spooler Buffering in CRE_Spool_Start</a>	6-30
<a href="#">Figure 6-4.</a>	<a href="#">Structure Allocation to Support Requesters Running as Process Pairs</a>	6-34
<a href="#">Figure 8-1.</a>	<a href="#">Strings in Memory Before Copying Source to Destination</a>	8-20
<a href="#">Figure 8-2.</a>	<a href="#">Strings in Memory After Copying Source to Destination</a>	8-20

## Tables

<a href="#">Table i.</a>	<a href="#">Language Manuals</a>	xvii
<a href="#">Table ii.</a>	<a href="#">System Programming Manuals</a>	xviii
<a href="#">Table iii.</a>	<a href="#">Program Development Manuals</a>	xviii
<a href="#">Table 1-1.</a>	<a href="#">CRE/RTL Files</a>	1-4
<a href="#">Table 1-2.</a>	<a href="#">Requirements for Running in the CRE</a>	1-5
<a href="#">Table 1-3.</a>	<a href="#">Language Support in TNS CRE and native CRE</a>	1-8
<a href="#">Table 2-1.</a>	<a href="#">CRE Services in the Guardian Environment</a>	2-5
<a href="#">Table 2-2.</a>	<a href="#">CRE Services in the OSS Environment</a>	2-5
<a href="#">Table 2-3.</a>	<a href="#">PARAMs Processed by the CRE</a>	2-11
<a href="#">Table 2-4.</a>	<a href="#">CRE Initialization Errors</a>	2-14
<a href="#">Table 2-5.</a>	<a href="#">Heap-Management Attributes for the High-Performance Heap Manager</a>	2-46
<a href="#">Table 2-6.</a>	<a href="#">TNS CRE Services Available in the OSS and Guardian Environments</a>	2-56
<a href="#">Table 2-7.</a>	<a href="#">Native CRE Services Available in the OSS and Guardian Environments</a>	2-56
<a href="#">Table 2-8.</a>	<a href="#">TNS CRE Standard Functions Available in the OSS and Guardian Environments</a>	2-57
<a href="#">Table 2-9.</a>	<a href="#">Native CRE Standard Functions Available in the OSS and Guardian Environments</a>	2-57

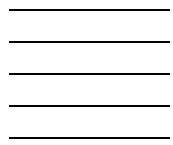
**Tables (continued)**

<a href="#">Table 2-10.</a>	<a href="#">Status Codes Returned by CRE Functions That Support Process Pairs</a>	<a href="#">2-61</a>
<a href="#">Table 3-1.</a>	<a href="#">ENV Options and the Availability of Run-Time Libraries</a>	<a href="#">3-2</a>
<a href="#">Table 3-2.</a>	<a href="#">ENV Options and the Availability of Language Features</a>	<a href="#">3-2</a>
<a href="#">Table 3-3.</a>	<a href="#">Determining Which ENV Options to Use</a>	<a href="#">3-2</a>
<a href="#">Table 3-4.</a>	<a href="#">Recognized and Default ENV Options</a>	<a href="#">3-3</a>
<a href="#">Table 3-5.</a>	<a href="#">Common Run-Time Environment Data Blocks</a>	<a href="#">3-5</a>
<a href="#">Table 3-6.</a>	<a href="#">Binder Grouping of ENV Directive Parameters</a>	<a href="#">3-5</a>
<a href="#">Table 3-7.</a>	<a href="#">Run-Time Environment Resulting From Binding Modules</a>	<a href="#">3-6</a>
<a href="#">Table 3-8.</a>	<a href="#">Locations of C-Series and D-Series TNS Run-Time Libraries</a>	<a href="#">3-7</a>
<a href="#">Table 4-1.</a>	<a href="#">env Pragma and the Availability of Features</a>	<a href="#">4-2</a>
<a href="#">Table 5-1.</a>	<a href="#">Comparison of CRE and CLU Functions</a>	<a href="#">5-1</a>
<a href="#">Table 5-2.</a>	<a href="#">SMU Functions</a>	<a href="#">5-3</a>
<a href="#">Table 5-3.</a>	<a href="#">Pre-D20 and Current SMU Functions</a>	<a href="#">5-4</a>
<a href="#">Table 5-4.</a>	<a href="#">Using SMU Functions</a>	<a href="#">5-6</a>
<a href="#">Table 5-5.</a>	<a href="#">Startup Message Parts</a>	<a href="#">5-6</a>
<a href="#">Table 5-6.</a>	<a href="#">ASSIGN Message Parts</a>	<a href="#">5-6</a>
<a href="#">Table 5-7.</a>	<a href="#">Retrievable Message Parts</a>	<a href="#">5-8</a>
<a href="#">Table 6-1.</a>	<a href="#">CRE Environment Functions</a>	<a href="#">6-1</a>
<a href="#">Table 6-2.</a>	<a href="#">File-Sharing Functions</a>	<a href="#">6-4</a>
<a href="#">Table 6-3.</a>	<a href="#">Language-Specific Constructs for Standard Files</a>	<a href="#">6-16</a>
<a href="#">Table 6-4.</a>	<a href="#">\$RECEIVE Functions</a>	<a href="#">6-31</a>
<a href="#">Table 6-5.</a>	<a href="#">Receive File Message Report Names</a>	<a href="#">6-36</a>
<a href="#">Table 6-6.</a>	<a href="#">Using Report^flags</a>	<a href="#">6-37</a>
<a href="#">Table 6-7.</a>	<a href="#">Default Values for Options and Completion Code</a>	<a href="#">6-44</a>
<a href="#">Table 6-8.</a>	<a href="#">Exception-Handling Functions</a>	<a href="#">6-45</a>
<a href="#">Table 7-1.</a>	<a href="#">TNS CRE Standard Math Functions</a>	<a href="#">7-2</a>
<a href="#">Table 7-2.</a>	<a href="#">Native CRE Standard Math Functions</a>	<a href="#">7-3</a>
<a href="#">Table 7-3.</a>	<a href="#">Sixty-Four-Bit Logical Operations</a>	<a href="#">7-23</a>
<a href="#">Table 7-4.</a>	<a href="#">Decimal Conversion Functions</a>	<a href="#">7-25</a>
<a href="#">Table 7-5.</a>	<a href="#">Sign Types</a>	<a href="#">7-29</a>
<a href="#">Table 8-1.</a>	<a href="#">TNS CRE String Functions</a>	<a href="#">8-1</a>
<a href="#">Table 8-2.</a>	<a href="#">Native CRE String Functions</a>	<a href="#">8-3</a>
<a href="#">Table 8-3.</a>	<a href="#">TNS CRE Memory Block Functions</a>	<a href="#">8-33</a>
<a href="#">Table 8-4.</a>	<a href="#">Native CRE Memory Block Functions</a>	<a href="#">8-33</a>
<a href="#">Table 9-1.</a>	<a href="#">SMU Functions</a>	<a href="#">9-18</a>
<a href="#">Table 10-1.</a>	<a href="#">C Message Mapping</a>	<a href="#">10-26</a>
<a href="#">Table 10-2.</a>	<a href="#">COBOL85 Message Mapping</a>	<a href="#">10-26</a>

**Tables** (continued)

<a href="#">Table 10-3.</a>	<a href="#">FORTRAN Message Mapping</a>	10-31
<a href="#">Table A-1.</a>	<a href="#">Integer Types, Part 1</a>	A-1
<a href="#">Table A-2.</a>	<a href="#">Integer Types, Part 2</a>	A-3
<a href="#">Table A-3.</a>	<a href="#">Floating, Fixed, and Complex Types</a>	A-4
<a href="#">Table A-4.</a>	<a href="#">Character Types</a>	A-5
<a href="#">Table A-5.</a>	<a href="#">Structured, Logical, Set, and File Types</a>	A-6
<a href="#">Table A-6.</a>	<a href="#">Pointer Types</a>	A-7





# What's New in This Guide

## Guide Information

### Abstract

This manual describes the Common Run-Time Environment (CRE), a set of run-time services that enable system and application programmers to write mixed-language programs. In the TNS environment, the TNS CRE run-time services support C, COBOL, FORTRAN, and TAL language programs. In the native environment, the native CRE run-time services support C, C++, COBOL, and pTAL language programs.

### Product Version

CRE G09, CRE H01

### Supported Release Version Updates (RVUs)

This manual supports G06.25 and all subsequent G-series RVUs and H06.03 and all subsequent H-series RVUs until otherwise indicated by its replacement publication.

Part Number	Published
528146-004	February 2006

### Document History

Part Number	Product Version	Published
528146-003	CRE G09, CRE H01	July 2005
528146-004	CRE G09, CRE H01	February 2006

## New and Changed Information

This manual contains information about some of the following G-series development tools. On HP Integrity NonStop NS-series servers, these tools are supported only in H06.05 and subsequent H-series RVUs.

- TNS/R native C compiler
- TNS/R native C++ compiler
- TNS/R native C++ runtime library version 2
- SQL/MP for TNS/R native C
- SQL/MP Compilation Agent
- NMCOBOL compiler or `nmcobol` front end
- `ld`
- `nld`
- `noft`
- NS/R pTAL

If you are running H06.03 or H06.04 RVUs, continue to use the HP Enterprise Toolkit--NonStop Edition or HP NonStop S-series servers with these development tools.

This manual contains the following changes from its G-series predecessor:

- Certain information has been restructured for clarity. For example, [Table 1-2](#) on page 1-5 replaced several paragraphs of text, and the message numbers in [Section 10, Run-Time Diagnostic Messages](#) were removed from the table of contents.
- H-series information was added, including:
  - Declaration source file and library file names
  - Linker and loader information
  - Wide-character functions `wmemchr ( )`, `wmemcmp ( )`, `wmemcpy ( )`, `wmemmove ( )`, and `wmemset ( )`
- G-series information about dynamic-link libraries was added for convenience when maintaining TNS/R native files on H-series systems.
- Information about C-series systems and compiler features was removed.
- A sentence excluding DCE and Java from use of the high-performance heap manager was removed in response to Genesis Solution 10-041018-0808.
- The appendix describing Guardian and logical file names and TACL commands was removed. You can find more current information in the *Guardian Procedure Calls Reference Manual*, the *Guardian Programmer's Guide*, and the *TACL Reference Manual*.



# About This Guide

The Common Run-Time Environment (CRE) is a set of services used by the HP compiler language run-time libraries. The CRE makes it easier to write mixed-language programs.

The *Common Run-Time Environment (CRE) Programmer's Guide* describes:

- Run-time environments and how they affect programs.
- Services provided by the CRE, such as program initialization and termination, file sharing, trap handling, and heap allocation.
- How to compile and bind programs that use the CRE.
- CRE and Common Language Utility (CLU) library run-time functions and diagnostic messages.

This guide contains only G-series and H-series product information needed to modify the use of CRE. G-series information is included because G-series compilers and utilities are available on H-series systems. Although not discussed in this guide, TNS object code created by C-series C compilers or the D-series Pascal compiler can be accelerated and run on an H-series system if it already executes successfully on G-series systems. For actions related to making such code work with CRE, refer to the latest D-series edition of this guide and to the final editions of the manuals for those compilers; neither the C-series compilers nor the Pascal compiler are available on H-series systems.

Unless otherwise indicated in the text, discussions of native-mode behavior, processes, and so forth apply to both the TNS/R native code that runs on G-series systems and to the TNS/E native code that runs on H-series systems. Discussions of TNS or accelerated code behavior in the OSS environment apply only to G-series systems; H-series systems do not support TNS or accelerated code execution in the OSS environment.

There are three HP compilers for the COBOL language, invoked by six separate commands. All these compilers implement the 1985 standard known as COBOL85 and are released as the product HP COBOL85 for NonStop Servers:

- The TNS compiler COBOL85 command in the Guardian environment or the `cobol` command in the G-series Open System Services (OSS) environment
- The TNS/R native compiler NMCOBOL command in the Guardian environment or the `nmcobol` command in the OSS environment
- The TNS/E native compiler ECOBOL command in the Guardian environment or the `ecobol` command in the OSS environment

An older COBOL language standard, the 1974 version, is no longer supported on HP NonStop servers. Neither that COBOL 74 product nor its Guardian-environment compiler are included in discussions in this guide unless specifically mentioned.

The SCREEN COBOL product is also excluded from COBOL discussions in this guide unless specifically mentioned.

To reduce confusion between the current product name and compiler-specific or platform-specific behaviors, this guide now uses “COBOL” to refer to the language when a more specific distinction is not needed or is obvious from context. “COBOL85” is used only to refer to the HP product or to the TNS compiler in the Guardian environment. “TNS COBOL” therefore includes the COBOL85 compiler, the OSS `cobol` compiler, and the TNS version of the COBOL run-time library, while “native COBOL” can refer to either the TNS/R or TNS/E compiler.

## Audience

This guide is intended for system and application programmers familiar with HP NonStop servers and the HP NonStop operating system. Readers of this guide should be familiar with the reference manuals and programmer’s guides of the languages in which their programs are written.

## Organization

This guide covers these topics:

[Section 1, Introducing the CRE](#), describes run-time environments and how they affect programs.

[Section 2, CRE Services](#), describes the services provided by the CRE, such as program initialization and termination, file sharing, trap handling, heap allocation and heap management.

[Section 3, Compiling and Binding Programs for the TNS CRE](#), shows how to compile and bind programs to run in the TNS CRE.

[Section 4, Compiling and Linking Programs for the Native CRE](#), shows how to compile programs to run in the native CRE.

[Section 5, Using the Common Language Utility \(CLU\) Library](#), explains how to use the functions provided by the Common Language Utility (CLU) library.

[Section 6, CRE Service Functions](#), describes CRE functions that handle file sharing, \$RECEIVE, program termination, and traps.

[Section 7, Math Functions](#), describes the standard math, 64-bit logical, and decimal-conversion functions provided by the CRE.

[Section 8, String and Memory Block Functions](#), describes the string and memory-block functions provided by the CRE.

[Section 9, Common Language Utility \(CLU\) Library Functions](#), describes the syntax of the functions in the Common Language Utility (CLU) library.

[Section 10, Run-Time Diagnostic Messages](#), lists the cause, effect, and recovery for each of the error messages emitted by programs running in the CRE.

[Appendix A, Data Type Correspondence](#), shows the relationships between data types among different programming languages.

## Additional Information

**Table i. Language Manuals**

Manual	Description
<i>C/C++ Programmer's Guide</i>	Describes the syntax and semantics of HP C and C++ for NonStop servers. Provides guidelines on writing C and C++ programs for NonStop servers.
<i>COBOL Manual for TNS and TNS/R Programs,</i> <i>COBOL Manual for TNS/E Programs</i>	Describes the syntax and semantics of HP COBOL for NonStop servers. Provides task-oriented information useful to COBOL programmers.
<i>Data Definition Language (DDL) Reference Manual</i>	Describes the syntax and semantics of the Data Definition Language.
<i>FORTTRAN Reference Manual</i>	Describes the syntax and semantics of HP FORTRAN.
<i>Guardian TNS C Library Calls Reference Manual</i>	Describes the syntax and semantics of C library calls for TNS and accelerated programs in the Guardian environment.
<i>Open System Services Library Calls Reference Manual</i>	Describes the syntax and semantics of C library calls for the HP NonStop Open System Services environment.
<i>pTAL Conversion Guide</i>	Explains how to convert TAL source code modules to be compiled with the pTAL compiler.
<i>pTAL Guidelines for TAL Programmers</i>	Provides guidelines for writing programs in TAL to change existing TAL code or write new TAL code that converts to pTAL.
<i>pTAL Reference Manual</i>	Describes the syntax and semantics of the Portable Transaction Application Language (pTAL).
<i>TAL Programmer's Guide,</i> <i>TAL Programmer's Guide</i> <i>Data Alignment Addendum</i>	Provides task-oriented information useful to TAL programmers.
<i>TAL Reference Manual</i>	Describes the syntax and semantics of TAL.

**Table ii. System Programming Manuals**

<b>Manual</b>	<b>Description</b>
<i>Guardian Application Conversion Guide</i>	Explains how to convert C-series file system procedure calls to use the extended features of the D-series file system procedure calls.
<i>Guardian Programmer's Guide</i>	Explains how to use the Guardian programmatic interface of the NonStop operating system in the Guardian environment.
<i>Guardian Procedure Calls Reference Manual</i>	Describes the syntax and programming considerations for using Guardian system procedures.
<i>Guardian Procedure Errors and Messages Manual</i>	Describes error codes, error lists, system messages, and trap numbers for system procedures.
<i>Open System Services Programmer's Guide</i>	Explains how to use the OSS programmatic interface of the NonStop operating system in the OSS environment.
<i>Open System Services Shell and Utilities Reference Manual</i>	Provides a complete description of OSS commands and utilities.
<i>Open System Services System Calls Reference Manual</i>	Describes the syntax and programming considerations for using OSS library calls.
<i>TACL Reference Manual</i>	Describes the syntax for specifying TACL command interpreter commands.
<i>TNS/E Native Application Migration Guide</i>	Gives guidelines for migrating applications from the TNS or TNS/R environment to the TNS/E native environment.
<i>TNS/R Native Application Migration Guide</i>	Gives guidelines for migrating applications from the TNS environment to the TNS/R native environment.

**Table iii. Program Development Manuals** (page 1 of 2)

<b>Manual</b>	<b>Description</b>
<i>Binder Manual</i>	Explains how to bind object files using Binder.
<i>Crossref Manual</i>	Explains how to collect cross-reference information using Crossref.
<i>Debug Manual</i>	Explains how to debug programs using the Debug machine-level interactive debugger.
<i>eld Manual</i>	Explains how to link TNS/E environment PIC object files using the eld utility.
<i>Inspect Manual</i>	Explains how to debug programs using the Inspect source-level and machine-level interactive debugger.
<i>ld Manual</i>	Explains how to link TNS/R environment PIC object files using the ld utility and to load object files using the rld utility.

**Table iii. Program Development Manuals** (page 2 of 2)

Manual	Description
<i>Native Inspect Manual</i>	Describes how to use Native Inspect, the built-in conversational symbolic debugger on TNS/E systems. Native Inspect is based on the UNIX debuggers GDB and WDB.
<i>nld Manual</i>	Explains how to link TNS/R environment non-position-independent-code (non-PIC) object files using the nld utility.
<i>rld Manual</i>	Explains how to load object files using the rld utility.
Visual Inspect online help	Describes how to use Visual Inspect, a graphical symbolic debugging product that provides powerful data display, application navigation, and multi-program support capabilities. Visual Inspect consists of a Windows client and a NonStop host-based server.

## 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 is a hyperlink to [Notation for Messages](#) on page xxi.

## General Syntax Notation

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

**UPPERCASE LETTERS.** Uppercase letters indicate keywords and reserved words; enter 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; enter 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 may 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 may 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 entered 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 enter 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 the following example, there are no spaces 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

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

**Bold Text.** Bold text in an example indicates user input entered 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 might 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 might 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*

## Change Bar Notation

A change bar (as shown to the right of this paragraph) indicates a difference between this edition of this guide and the preceding edition. Change bars highlight new or revised information.



# 1

## Introducing the CRE

This section describes the differences between using the Common Run-Time Environment (CRE) in the Guardian and the HP NonStop Open System Services (OSS) environments, and in the TNS, TNS/R, and TNS/E native environments. This section contains the following topics:

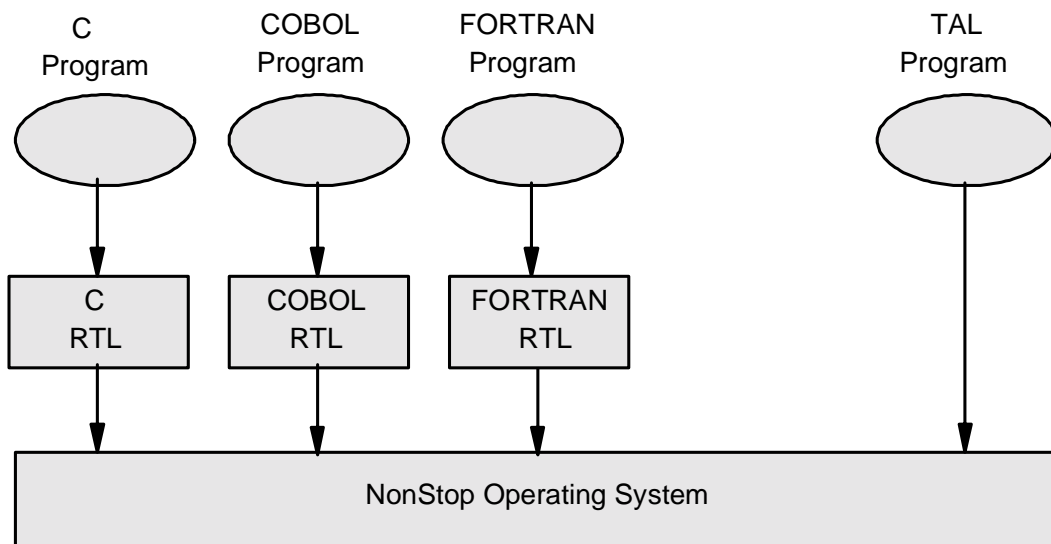
- [Mixed-Language Programming Without the CRE](#) on page 1-1
- [What Is the CRE?](#) on page 1-2
- [Selecting a Run-Time Environment](#) on page 1-5
- [Advantages of Using the CRE](#) on page 1-8

## Mixed-Language Programming Without the CRE

By default, the standards-based TNS programming languages each have their own run-time environments defined by their respective run-time libraries (see [Figure 1-1](#)). Each run-time library includes the math, string, heap management, trap and exception handling, and I/O management functions that define each language-specific run-time environment. The run-time libraries call system routines to obtain run-time services.

The language-specific run-time environments are different from one another and often incompatible. Mixed-language programs running in these environments are limited in their ability to use all of the features of each language and to share data between routines written in different languages. This incompatibility severely limits the potential for creating useful mixed-language programs.

**Figure 1-1. Language-Specific Run-Time Environments**



VST 101.VSD

---

**Note.** For simplicity, Figure 1-1 does not show the TAL run-time library or explicit calls to system routines in non-TAL programs. Only TAL programs with embedded SQL statements use the TAL run-time library when they run without the CRE.

---

For a mixed-language program running in a language-specific run-time environment, the program's main routine determines the run-time environment for all routines in the program. The run-time libraries do not coordinate with each other.

For example, the main routine might establish a table of run-time error messages and include error-reporting routines that require the table to be at a fixed location. It might also allocate a set of data blocks for keeping track of environment information for the set of files used by the user-written routines. Error reporting and file I/O operations follow the rules and schemes of the run-time environment established by the main routine's run-time library. The main routine can safely call a routine written in another language only if the called routine does not perform any operations that conflict with the rules and schemes of the run-time environment of the main routine. The called routine cannot perform operations that depend on the services of the called routine's own run-time environment, such as error reporting and file I/O operations.

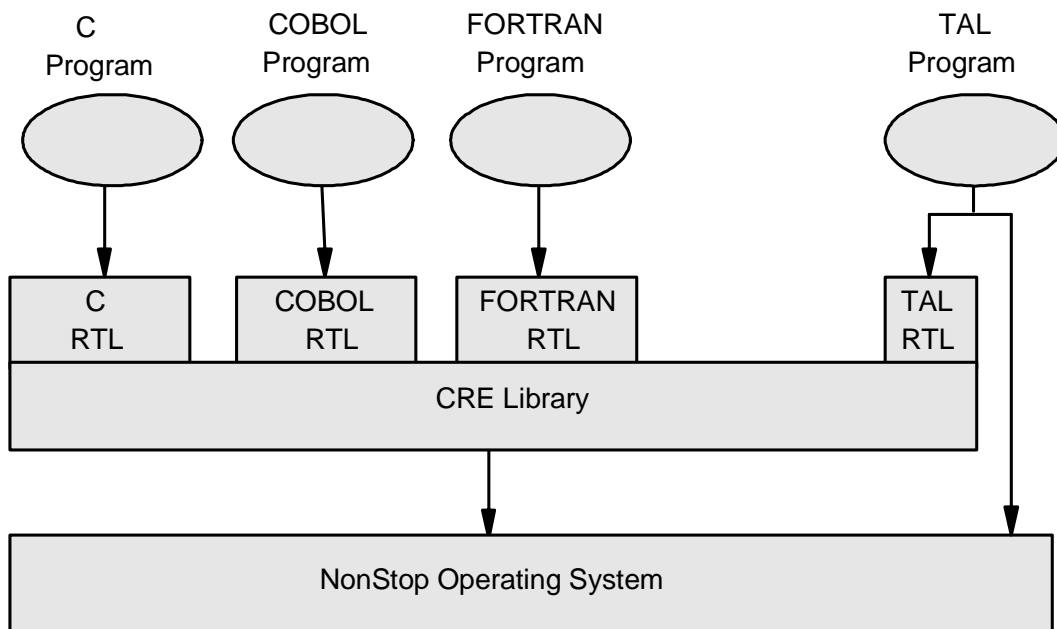
## What Is the CRE?

The Common Run-Time Environment (CRE) eliminates the problems cited in the previous subsection. Programs running in the OSS environment or the Guardian environment can use the services of the CRE, as shown in the following figures.

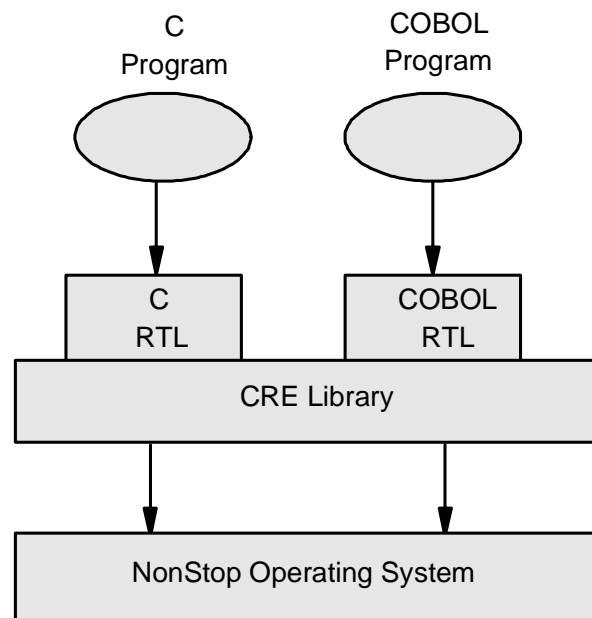
---

**Figure 1-2. The Common Run-Time Environment in the Guardian Environment**

---



VST 102.VS  
D

**Figure 1-3. The Common Run-Time Environment in the OSS Environment**

VST 103.VSD

The CRE provides several products with similar services:

- TNS CRE/RTL (product number T9280) supports programs running in the TNS environment
- Native CRE/RTL (product number T8431) supports programs running in the TNS/R native environment.
- NonStop CRE/RTL (product number T1269) supports programs running in the TNS/R or TNS/E native environments.

The TNS CRE is configured into the system library. The TNS/R native CRE is a public shared run-time library or hybrid dynamic-link library (DLL). The TNS/E native CRE is a DLL. [Table 1-1](#) on page 1-4 describes the files that make up the CRE/RTL products.

**Table 1-1. CRE/RTL Files**

<b>File name</b>	<b>Default Location</b>	<b>Description</b>
CLUDECS	\$SYSTEM.SYSTEM	Source file containing TNS CRE Saved Message Utility (SMU) functions, data, and data structure declarations in TAL.
CLURDECS	\$SYSTEM.SYSTEM	Source file containing TNS/R and TNS/E native CRE Saved Message Utility (SMU) functions, data, and data structure declarations in pTAL.
CREDECS	\$SYSTEM.SYSTEM	Source file containing TNS CRE functions, data, and data structure declarations in TAL.
CRERDECS	\$SYSTEM.SYSTEM	Source file containing TNS/R and TNS/E native CRE functions, data, and data structure declarations in pTAL.
TALLIB	\$SYSTEM.SYSTEM	Linkable code file containing the TNS CRE TAL initialization function definition.
RTLDECS	\$SYSTEM.SYSTEM	Source file containing TNS CRE RTL functions, data, and data structure declarations in TAL.
RTLREDECS	\$SYSTEM.SYSTEM	Source file containing TNS/R and TNS/E native CRE RTL functions, data, and data structure declarations in pTAL.
ZCRESRL	\$SYSTEM.SYS <sub>nn</sub>	Shared run-time library or hybrid DLL executable code file containing all TNS/R native CRE functions and data definitions.
ZCREDLL	\$SYSTEM.ZDLL <sub>nnn</sub>	DLL containing all TNS/E native CRE functions and data definitions.

The CRE coordinates many run-time tasks on behalf of the run-time libraries, thus providing a common environment for all routines in a program, regardless of language. The CRE provides services that significantly enhance your ability to create mixed-language programs.

The following table shows the services the CRE provides in the OSS and Guardian environments:

<b>Feature</b>	<b>OSS Environment</b>	<b>Guardian Environment</b>
Shared access to standard files (standard input, standard output, and standard log)	Available independent of the CRE	Provided by the CRE
Shared access to a common user heap	Provided by the CRE	Provided by the CRE
Management of \$RECEIVE	Provided by the CRE (for server management only)	Provided by the CRE

Feature	OSS Environment	Guardian
Management of process initialization and termination	Provided by the CRE	Provided by the CRE
Management of checkpoint and restart activities for process pairs	N.A.	Provided by the CRE
Uniform handling of exceptions	N.A.	Provided by the CRE
Uniform format and content of diagnostic messages	Provided by the CRE	Provided by the CRE

## Selecting a Run-Time Environment

[Table 1-2](#) on page 1-5 lists the requirements for running a program in the CRE.

**Table 1-2. Requirements for Running in the CRE** (page 1 of 3)

Primary language or compiler used for the program	Need to make source changes?	Must be compiled to run in the CRE?	Can use system procedure calls for resources that are not shared?	What changes in execution?	Can contain a main routine?
C	No, if the routine uses only operating system and language constructs	Yes	Yes	Only text and format of some run-time diagnostic messages, which come from the CRE error-reporting functions	Yes
C++	No, if the routine uses only operating system and language constructs	Yes.	Yes.	No.	Yes.

**Table 1-2. Requirements for Running in the CRE** (page 2 of 3)

<b>Primary language or compiler used for the program</b>	<b>Need to make source changes?</b>	<b>Must be compiled to run in the CRE?</b>	<b>Can use system procedure calls for resources that are not shared?</b>	<b>What changes in execution?</b>	<b>Can contain a main routine?</b>
COBOL85 or <code>cobol</code> (TNS)	No, if the routine uses only operating system and language constructs.	Yes, by specifying the ENV COMMON compiler directive.	Yes.	No.	Yes.
ECOBOL or <code>ecobol</code> (TNS/E)	No, if the routine uses only operating system and language constructs	No, but ENV COMMON compiler directive is default	Yes	No	Yes
NMCOBOL or <code>nmcobol</code> (TNS/R)	No, if the routine uses only operating system and language constructs	No, but ENV COMMON compiler directive is default	Yes	No	Yes
FORTRAN (TNS)	No, if the routine uses only operating system and language constructs	Yes, by specifying the ENV COMMON compiler directive	Not applicable	Only text and format of some run-time diagnostic messages, which come from the CRE error-reporting functions	Yes
EpTAL (TNS/E)	Yes; calls from system routines to CRE library functions to perform actions on those objects, such as the standard files, that you want to share or coordinate with other languages	No	Yes	No	No



**Table 1-2. Requirements for Running in the CRE** (page 3 of 3)

Primary language or compiler used for the program	Need to make source changes?	Must be compiled to run in the CRE?	Can use system procedure calls for resources that are not shared?	What changes in execution?	Can contain a main routine?
pTAL (TNS/R)	Yes; calls from system routines to CRE library functions to perform actions on those objects, such as the standard files, that you want to share or coordinate with other languages	No	Yes	No	No
TAL (TNS)	Yes; calls from system routines to CRE library functions to perform actions on those objects, such as the standard files, that you want to share or coordinate with other languages	Only if a compiler directive is specified	Yes	No	Yes, if changed to make calls to specific CRE library initialization and termination functions

Refer to [Section 3, Compiling and Binding Programs for the TNS CRE](#), and [Section 4, Compiling and Linking Programs for the Native CRE](#), for details. Refer to [Writing TAL Routines That Use the TNS CRE](#) on page 2-6 for additional restrictions for TAL routines running in the TNS CRE.

pTAL routines linked into a program with a native C or native COBOL main routine can run in the CRE. (The native C or native COBOL main routine ensures that the correct CRE library initialization and termination functions are called.) Refer to [Writing TAL Routines That Use the TNS CRE](#) on page 2-6 for additional restrictions for pTAL routines running in the native CRE.

[Table 1-3](#) on page 1-8 compares the languages supported by the TNS CRE and the TNS/R or TNS/E native CRE.

**Table 1-3. Language Support in TNS CRE and native CRE**

Application Language	TNS CRE	Native CRE
COBOL	Yes (using the COBOL85 or <code>cobol</code> compiler)	Yes (using the ECOBOL, <code>ecobol</code> , <code>nmcobol</code> , or NMCOBOL compiler)
FORTTRAN	Yes	No
C	Yes	Yes
C++	Yes	Yes
TAL	Yes	No
pTAL	No	Yes

**Note.** The CRE does not support mixed TNS and native programs (that is, programs that contain both TNS object code/accelerated object code and TNS/R or TNS/E native object code).

## Advantages of Using the CRE

In the Guardian environment, the CRE has these advantages:

- Removes many restrictions on mixed-language programming. For example, in the Guardian environment all routines in a program can share access to the standard files (standard input, standard output, and standard log) and `$RECEIVE`, not just those written in the same language. (In the OSS environment, the file system supports standard files explicitly.)
- Allows a routine to be written in the language best suited for its task. For example:
  - A program can have its file operations written in COBOL and its list management in TAL or C.
  - A software development group might have a set of standard FORTRAN routines that it uses to build its TAL or COBOL programs. The CRE makes these routines available to all programs, regardless of the language.
- Allows access to functions that are not defined as part of its own language or run-time library. The CRE functions can be called from any language.
- Enables you to convert applications from one programming language into another programming language in phases. For example, suppose you need to convert an application from TAL into C. You can divide the routines in the TAL application into functional sets. As you convert each set of routines to C, you can test each set by binding the TAL and C routines into one program.
- Allows easier porting of programs to HP NonStop servers. Not only does the CRE make it easier to port mixed-language programs, but you can rewrite performance-critical routines in the language best suited for running on NonStop servers.

# 2 CRE Services

This section describes the services and resources managed by the CRE. In this section, *CRE* refers to both the TNS environment and native environments. Where there are differences, *TNS CRE* and *TNS/R native CRE* or *TNS/E CRE* are used.

Your C, COBOL, and FORTRAN routines in the TNS environment, and your C, C++, COBOL, and pTAL routines in the native environments access most CRE services transparently to you through their run-time libraries. Each run-time library translates requests expressed in the syntax of its language into calls to CRE library functions or system procedures. When you use high-level language constructs to read or write a standard file, or to call a math or string function, each run-time library calls CRE library functions that perform the requested operation. Your routines specify the parameters of each operation using the syntax of the routine's language; you never need to know the specifics of the CRE library functions.

Most of the topics described in this section are informational. You must read this section only if you are writing TAL routines and you want to share access to services and resources managed by the TNS CRE with routines written in other languages. If you are writing routines to compile with the C, TNS COBOL, and FORTRAN compilers, you do not need to know most of the details described in this section.

Use the following table to determine which subsections to read:

Read the Following Subsection:	In order to:
<a href="#">Comparing the CRE in the OSS and Guardian Environments</a> on page 2-2	Understand the differences between CRE services in the OSS and Guardian environments
<a href="#">Writing TAL Routines That Use the TNS CRE</a> on page 2-6	Write TAL routines that run in the TNS CRE
<a href="#">Writing pTAL Routines That Use the Native CRE</a> on page 2-8	Write pTAL language routines that run in the native CRE
<a href="#">Program Initialization</a> on page 2-9	Learn the steps that the CRE initialization function follows to initialize your program
<a href="#">Program Termination</a> on page 2-15	Learn the steps that the CRE termination function follows to terminate your program
<a href="#">Sharing Standard Files</a> on page 2-17	Share access to the standard files (standard input, standard output, and standard log) among routines written in different languages
<a href="#">Using \$RECEIVE</a> on page 2-34	Share access to \$RECEIVE among routines written in different languages (now available for both TNS C/C++ and native C/C++)
<a href="#">Using a Spooler Collector</a> on page 2-36	Access a spooler collector directly as a standard file
<a href="#">Memory Organization</a> on page 2-37	Access the run-time heap from C and TAL routines that run in the CRE

**Read the Following  
Subsection:****In order to:** (continued)[Using the Native Heap Managers](#)  
on page 2-43Learn how to use the native heap managers and how to  
query or set heap-management attributes[TNS CRE Traps and Exceptions](#)  
on page 2-47 and [Native CRE  
Signals and Exceptions](#) on  
page 2-55Learn how the CRE handles traps, signals, and  
exceptions[Reporting CRE Errors in the OSS  
Environment](#) on page 2-54Understand how the CRE determines where it writes  
error messages in the OSS environment[Using Standard Functions](#) on  
page 2-56

Call CRE library functions explicitly from your program

[Using Process Pairs](#) on  
page 2-59

Learn how the CRE manages process pairs

[Using the Inspect, Native Inspect,  
and Visual Inspect Symbolic  
Debuggers With CRE Programs](#)  
on page 2-62Use the corresponding symbolic debugger product to  
locate where a program is overwriting CRE data[Circumventing the CRE](#) on  
page 2-66Use system procedures to manipulate services  
managed by the CRE

Note that the CRE does not support Event Management Service (EMS) events.

## Comparing the CRE in the OSS and Guardian Environments

Many of the services provided by the CRE in the Guardian environment are not needed or are not meaningful in the OSS environment. This subsection summarizes the differences between CRE services in the OSS and Guardian environments for each of the following topics:

- [Standard Files](#) on page 2-2
- [\\$RECEIVE](#) on page 2-3
- [Memory Organization](#) on page 2-3
- [Traps and Exceptions](#) on page 2-3
- [Program Initialization](#) on page 2-4
- [Program Termination](#) on page 2-4
- [Error Reporting](#) on page 2-5
- [Standard Functions](#) on page 2-5
- [CRE Services](#) on page 2-5
- [Process Pairs](#) on page 2-6 (fault-tolerant programming)

### Standard Files

In the Guardian API or environment, the CRE supports three standard files (standard input, standard out, and standard log), and supports routines that enable program

modules to share standard files (standard out and standard log) that write to spooler collectors.

In the OSS environment, the file system supports standard files—the CRE, therefore, does not provide such support. Language-specific run-time libraries call OSS routines to access standard files. Because the CRE does not support standard files in the OSS environment, it does not support the CRE routines for sharing spooled files.

## \$RECEIVE

\$RECEIVE is supported in both the TNS and native CRE environments. In the Guardian and OSS environments, programs receive the following types of messages from \$RECEIVE: system messages and messages sent by other processes. In the Guardian environment, processes also receive initialization messages (startup message, ASSIGN messages, and PARAM message) from \$RECEIVE. Processes running in the OSS environment do not receive initialization messages from \$RECEIVE.

## Memory Organization

The CRE manages a user heap for programs running in either the OSS or Guardian environments.

For a given heap size, a program running in the OSS environment might have a different amount of available space than the same program running in the Guardian environment.

## Traps and Exceptions

For Guardian processes running in the TNS environment, the CRE enables a trap handler during program initialization and manages exception handling during program execution. Programs can use the Guardian ARMTRAP procedure to detect and process exceptions.

Guardian C programs use the signal feature to detect exceptions, rather than traps.

In the native environment, traps are replaced with signals.

For Guardian processes running in the native environment, the CRE provides a default signal handler for all signals whose default action is not signal ignore [SIGIGN]. The signal handler takes all appropriate actions, then terminates the process. The signal handler does not return control to its caller. Refer to the *Guardian Programmer's Guide* for more information on signals.

The OSS environment supports signals but does not support trap handlers. Therefore, the CRE does not regain control of a process if an exception occurs. In most cases, the OSS environment makes error information available to running programs. The CRE does not provide a signal handler for OSS processes. See the *OSS Programmer's Guide* for more information about exception handling in the OSS environment.

In both the OSS and Guardian environments, the CRE validates the environment during program execution. If it detects that the environment is corrupted, it writes a message for the user to read, and terminates the program.

## Program Initialization

### TNS CRE

In the Guardian environment, the TNS CRE:

- Initializes its own state
- Calls an initialization routine for each language-specific run-time environment that is represented in the program's object file
- Establishes a trap handler
- Processes initialization messages including the startup message, ASSIGN message, and PARAM messages
- Initializes structures for standard files and for process pairs

In the OSS environment on G-series systems, the TNS CRE:

- Initializes its own state
- Calls an initialization routine for each language-specific run-time environment that is represented in the program's object file

### Native CRE

The native CRE performs only those common steps that pertain to C, C++, native-mode COBOL, and pTAL run-time initialization. Each language-specific run-time library causes its own initialization routine to be asserted.

In the Guardian environment, the native CRE:

- Initializes its own state
- Establishes a signal handler
- Processes initialization messages including the startup message, ASSIGN message, and PARAM messages
- Initializes structures for standard files and for process pairs

In the OSS environment, the native CRE initializes its own state, and each language-specific run-time library causes its own initialization routine to be asserted.

## Program Termination

The CRE calls a language-specific termination routine for each language represented in the program's object file and then calls a kernel routine to stop the process. The

native CRE calls a kernel routine that calls all shared run-time library termination routines, and then calls another kernel routine to stop the process.

In the Guardian environment, the CRE also ensures that buffers for standard files are empty and properly closed.

Error Reporting

In general, the CRE writes messages to the standard log file. The location to which the CRE writes messages, however, can be different in the OSS and Guardian environments. See [Standard Log](#) on page 2-33 for more details.

Standard Functions

The CRE provides many standard functions, including:

- Math functions, described in [Section 7, Math Functions](#)
- String functions, described in [Section 8, String and Memory Block Functions](#)
- Memory block functions, described in [Section 8, String and Memory Block Functions](#)

The native CRE library supports only standard functions required by the C and C++ run-time libraries.

CRE Services

Table 2-1. CRE Services in the Guardian Environment

Service	TNS environment	Native environment
Standard files	Yes	Yes
\$RECEIVE message processing	Yes	Yes
Exception handling	Traps	Signals
Process initialization	Yes	Yes
Process termination	Yes	Yes
Process pairs	Yes	Yes

Table 2-2. CRE Services in the OSS Environment (page 1 of 2)

Service	TNS environment	Native environment
Standard files	No	No
\$RECEIVE message processing	Limited	Limited
Exception handling	Signals	Signals

**Table 2-2. CRE Services in the OSS Environment** (page 2 of 2)

Service	TNS environment	Native environment
Process initialization	Yes	Yes
Process termination	Yes	Yes
Process pairs	No	No

**Note.** All CRE service functions are visible to all programs in the Guardian and OSS environments. The results of calling a function that is not defined in the current environment, however, are undefined.

The Saved Message Utility (SMU) functions of the Common Language Utility (CLU) library are available only in the Guardian environment.

The native CRE library does not support CLU library functions that locate and identify file connectors.

## Process Pairs

The term *process pairs* refers to the use of primary and backup processes in fault-tolerant programming. In the Guardian environment, both the TNS and native CRE can manage the backup process on behalf of the running program.

The OSS environment does not support process pairs (primary and backup). Therefore, all CRE services and functions associated with process pairs are undefined in the OSS environment.

## Writing TAL Routines That Use the TNS CRE

Unlike C, COBOL, and FORTRAN routines, TAL routines must call TNS CRE functions explicitly to access TNS CRE services because the TAL run-time library does not support file I/O, math, or string functions. This subsection describes considerations for TAL routines that run in the TNS CRE in the Guardian environment.

Follow these guidelines to write TAL routines that run in the TNS CRE:

- Use a TAL ENV directive to establish the correct environment for your TAL programs. A TAL program can run in the TNS CRE if the TAL main routine has the ENV COMMON attribute and other TNS CRE requirements are met. [Section 3, Compiling and Binding Programs for the TNS CRE](#), describes the TAL ENV directive. See also the *TAL Reference Manual*.
- Do not manipulate directly the upper 32K of the user data segment. The TNS CRE uses this space to store its data objects. Do not use memory addresses G[0] or G[1].
- Do not directly manipulate program resources that are likely to be shared with other languages.



- Call `TAL_CRE_INITIALIZER_` in the first statement of a TAL main procedure. `TAL_CRE_INITIALIZER_` is located in the TAL run-time library, `TALLIB` and the external declaration is located in `TALDECS`. `TAL_CRE_INITIALIZER_` invokes TNS CRE functions that initialize the TNS CRE. [Program Initialization](#) on page 2-4 describes the TNS CRE's initialization tasks. The *TAL Programmer's Guide* describes the interface to `TAL_CRE_INITIALIZER_`. Note that the TNS CRE does not create backup processes.
- Call `CRE_Terminator_`, not the `PROCESS_STOP_`, `STOP` or `ABEND` system procedures, for TAL routines that terminate program execution.
- Call system procedures to access resources that are not managed by the TNS CRE.
- Call system procedures to access resources that can be managed by the TNS CRE but that you do not want to share with routines written in other languages.
- Call the TNS CRE functions described in [Section 6, CRE Service Functions](#), to access resources managed by the TNS CRE that you want to share with routines written in other languages.

For example, call TNS CRE library functions to share access to the standard files or `$RECEIVE`. Note that the presence of these objects does not necessarily mean that your program shares these resource amongst routines written in multiple languages. If no other language is reading from or writing to `$RECEIVE`, then your TAL program can open, read, write, and close `$RECEIVE` using system procedures, rather than TNS CRE library functions.

For example, call the `PROCESS_CREATE_` system procedure in your TAL routine to create a new process. The TNS CRE does not provide a run-time function to create processes on behalf of your program.

- Source in the file `CREDECS`
- Do not use the `INITIALIZER` system procedure. Both the TNS CRE and the `INITIALIZER` procedure read system messages, including the startup message, the `PARAMs` message, and `ASSIGN` messages.

Programs that use sequential I/O (SIO) procedures (for example, `OPEN^FILE`, `READ^FILE`, `WRITE^FILE`, `CLOSE^FILE`, and so forth) are particularly likely to use the `INITIALIZER` system procedure.

You must resolve how to remove the call to the `INITIALIZER` procedure from your program.

The *Guardian Procedure Calls Reference Manual* describes the `INITIALIZER` procedure and each of the SIO procedures. The *Guardian Programmer's Guide* describes how to use the `INITIALIZER` system procedure with SIO procedures.

- If your TAL routines call standard math functions, the TAL routines must manage the trap enable bit of the TNS CRE environment register to control program

behavior if a math function detects an error. For more details, see [Traps and Exceptions](#) on page 2-3.

For more information on using TAL in mixed-language programs and writing TAL programs that use TNS CRE services, see the *TAL Programmer's Guide*.

## Writing pTAL Routines That Use the Native CRE

This subsection describes considerations for pTAL language routines that run in the CRE. Unlike C and C++ routines, pTAL routines must call native CRE functions explicitly to access native CRE services because pTAL does not support standard input, standard output, and standard error file I/O, math, or string functions.

pTAL cannot be used to create the main routine of a program that runs in the CRE. A C or native-mode COBOL main routine that calls a pTAL routine can be used to provide almost the same functionality.

A program with a pTAL main routine cannot link to a native user library that runs in the CRE. Thus, the native user library for such a program cannot be written in C, C++, or native-mode COBOL.

A program with a C or native-mode COBOL main routine can link to a native user library that runs in the CRE or follows the guidelines for CRE compliance. Thus, the native user library for such a program can be written in C, native-mode COBOL, or in pTAL that complies with the following guidelines:

- Do not directly manipulate program resources that are likely to be shared with other languages.
- Call `CRE_Terminator_`, not the `PROCESS_STOP_`, `STOP` or `ABEND` system procedures, for pTAL routines that terminate program execution.
- Call system procedures to access resources that are not managed by the CRE.
- Call system procedures to access resources that can be managed by the CRE, but that you do not want to share with routines written in other languages.
- Call the native CRE functions described in [Section 6, CRE Service Functions](#), to access resources managed by the native CRE that you want to share with routines written in other languages.
- Source in the files `CRERDECS`, `RTLREDECS`, or `CLURDECS` where applicable.
- Do not use the `INITIALIZER` system procedure. Both the CRE and the `INITIALIZER` procedure read system messages, including the startup message, the `PARAMs` message, and `ASSIGN` messages.

Programs that use sequential I/O (SIO) procedures (for example, `OPEN^FILE`, `READ^FILE`, `WRITE^FILE`, `CLOSE^FILE`, and so forth) are particularly likely to use the `INITIALIZER` system procedure.

You must resolve how to remove the call to the INITIALIZER procedure from your program.

The *Guardian Procedure Calls Reference Manual* describes the INITIALIZER procedure and each of the SIO procedures. The *Guardian Programmer's Guide* describes how to use the INITIALIZER system procedure with SIO procedures.

## Program Initialization

Your program begins execution when the operating system transfers control to your program's object code. Before executing the code that you wrote, however, the run-time library for your main routine initializes its run-time environment. For programs running in the CRE, initialization includes a call to a CRE initialization function.

In the Guardian environment, the CRE performs the tasks described in this subsection. In the OSS environment on G-series systems, the TNS CRE does not:

- Establish a trap/signal handler
- Process startup messages
- Initialize standard files
- Initialize process pair information

The CRE initialization function establishes the CRE's internal data structures, I/O model, and so forth, as well as shared facilities such as the user data heap, standard files (standard input, standard output, and standard log), and parameters that control process pairs. After the CRE has established its environment and set up shared facilities, it calls a language-specific initialization function for each language that is represented by a routine in your program, except TAL and pTAL. Each language-specific initialization function sets up its data structures and file I/O model for the language that it supports.

When the CRE completes initialization, it is set up to provide the services described in this section, and returns control to the run-time library that called it, namely, the run-time library for your main routine. The run-time library completes its own initialization and returns control to your main routine, which begins executing the instructions that you wrote.

## Designating a Main Routine

A program's main routine is the first routine to execute when the operating system passes control to your process. An object file must have exactly one main routine to be a runnable program. Each language supported by the CRE provides syntax to specify

a main routine. The following table shows an example of the syntax of a main routine, called MYMAIN, in each of the languages supported by the CRE.

Language	Examples of Main Routine Declarations
C	<pre>int main();    /* C main routine */ { ... };      /*   called "main"   */</pre>
COBOL	<pre>?MAIN mymain IDENTIFICATION DIVISION. PROGRAM-ID. mymain.</pre>
FORTRAN	<pre>PROGRAM mymain</pre>
TAL	<pre>PROC mymain MAIN;</pre>
pTAL	<pre>BEGIN ... END;</pre>

The reference manual for each language describes the syntax of a main routine.

## TNS CRE Initialization

During program initialization, the TNS CRE:

- Opens \$RECEIVE, processes the startup message, ASSIGN messages, and the PARAM message, and closes \$RECEIVE.
  - For C main routines, the CRE saves the names and values of all PARAMs and ASSIGNs and returns them to your program automatically.
  - For COBOL and FORTRAN main routines, the CRE saves the names and values and returns them to your program if you specified the SAVE directive for the main routine.
  - For a TAL main routine, the CRE saves the names and values and returns them to your program if you specified the message-saving parameter in the TAL\_CRE\_INITIALIZER\_ function.

The *TACL Reference Manual* describes how you specify ASSIGNs and PARAMs in TACL. The *Guardian Programmer's Guide* describes how you process startup messages, ASSIGNs, and PARAMs.

The CRE processes ASSIGN messages following the same rules as the INITIALIZER system procedure. See the *Guardian Procedure Calls Reference Manual* for details.

[Table 2-3](#) on page 2-11 shows the PARAMs that the CRE explicitly processes during initialization. The PARAM Values column shows the values that the CRE accepts for each PARAM. The CRE terminates your program if the value associated with any PARAM symbolic name is not valid. Termination occurs regardless of which languages are used to create program modules.

**Table 2-3. PARAMs Processed by the CRE** (page 1 of 2)

PARAM Name	PARAM Values	Default Value	Specifies:
BUFFERED-SPOOLING	ON OFF	ON	Whether buffered spooling is the default
DEBUG	ON OFF	OFF	How COBOL85 or <code>cobol</code> handles the DEBUG switch. See the <i>COBOL Manual for TNS and TNS/R Programs</i> .
EXECUTION-LOG	<i>filename</i>	N.A.	The file to which the CRE writes log messages. It can also be used to specify the standard input and standard output files. For more details, see <a href="#">Standard Log</a> on page 2-33, <a href="#">Standard Input</a> on page 2-29, and <a href="#">Standard Output</a> on page 2-31.
INSPECT	ON OFF	OFF	Whether the CRE invokes a debugger or terminates your program if certain language-specific run-time errors occur. Although this PARAM is called INSPECT, you can control whether your program invokes the default debugging utility or the symbolic debugging utility. For more details, see the DEBUG and PARAM commands in the <i>TACL Reference Manual</i> .
NONSTOP	ON OFF	ON	Whether your program can run as a process pair. To run as a process pair, your program must also specify the NONSTOP compiler directive, which is supported only by the TNS COBOL, TNS/R COBOL (software product revision T8107AAT or more recent), TNS/E COBOL, and FORTRAN compilers.

**Table 2-3. PARAMs Processed by the CRE** (page 2 of 2)

PARAM Name	PARAM Values	Default Value	Specifies:
PRINTER-CONTROL	<i>filename</i> <i>*.filename</i>		<p>That the operating system return control to your program if the printer specified by the value of the PRINTER-CONTROL PARAM is either not ready or out of paper. The semantics for PARAM value are:</p> <ul style="list-style-type: none"> <li>● <i>filename</i> matches a file in any program but cannot appear in more than one program.</li> <li>● <i>*.filename</i> matches a file in any program and can match files in more than one program</li> <li>● <i>progrname.filename</i> matches a filename only in the program <i>progrname</i>.</li> </ul>
SAVE-ENVIRONMENT	ON OFF	See next column	<p>Saves environment information, derived from PARAMs and startup message, into the <code>environ</code> array. By default, SAVE-ENVIRONMENT is ON if the main routine is written in C, and OFF if the main routine is not written in C. Use the SAVE-ENVIRONMENT PARAM to override these defaults. ASSIGNs, PARAMs, and startup messages are available in the Guardian environment but are not available in the OSS environment, regardless of the value of the SAVE-ENVIRONMENT PARAM.</p>
SWITCH-1 ... SWITCH-15	ON OFF	OFF	<p>The value of the corresponding switch. Switches can be tested only by TNS COBOL and FORTRAN programs. For details, see the <i>COBOL Manual for TNS and TNS/R Programs</i> or the <i>FORTRAN Reference Manual</i>.</p>

- Determines the name of your program's standard log. (Standard log is another name for what some languages call STDERR. Messages written to STDERR appear in standard log. Messages in standard log might be errors, warnings, or informational.) If the CRE cannot determine the name, it terminates your program. [Standard Files](#) on page 2-2 describes how the CRE determines the name of the file to open for standard log.
- Establishes a trap handler.
- Invokes an initialization function for each language in your program except TAL.

- Opens certain shared standard files (standard input, standard output, standard log), depending on the language of your main routine:
  - If your main routine is written in C, the CRE calls a C run-time library function to open all three standard files unless you have specified the `nostdfiles` pragma.
  - If your program is written in COBOL, FORTRAN, or TAL, the TNS CRE opens each standard file only when it receives an open request from your program.

## Native CRE Initialization

During program initialization in the Guardian environment, the native CRE performs the following:

- Processes the startup message, ASSIGN messages, and the PARAM message
- Saves the names and values of all PARAMs and ASSIGNs and returns them to your program automatically

The *TACL Reference Manual* describes how you specify ASSIGNs and PARAMs in TACL. The *Guardian Programmer's Guide* describes how you process startup messages, ASSIGNs, and PARAMs.

The CRE processes ASSIGN messages following the same rules as the INITIALIZER system procedure. See the *Guardian Procedure Calls Reference Manual* for details.

[Table 2-3](#) on page 2-11 shows the PARAMs that the CRE explicitly processes during initialization. The PARAM Values column of the table shows the values that the CRE accepts for each PARAM. The CRE terminates your program if the value associated with any PARAM symbolic name is not valid.

- Determines the name of your program's standard log. (Standard log is another name for what some languages call STDERR. Messages written to STDERR appear in standard log. Messages in standard log might be errors, warnings, or informational.) If the CRE cannot determine the name, it terminates your program. [Standard Files](#) on page 2-2 describes how the CRE determines the name of the file to open for standard log.
- Establishes a signal handler.
- Invokes an HP NonStop operating system initialization function.
- Opens certain shared standard files (standard input, standard output, standard log), depending on the language of your main routine:
  - If your main routine is written in C, the CRE calls a C run-time library function to open all three standard files unless you have specified the `nostdfiles` pragma.

## Initialization Errors

Several kinds of errors can occur during CRE initialization. If an error occurs before the CRE has set up its environment, the CRE invokes `PROCESS_STOP_` and specifies option `ABEND` with the appropriate error text, as listed in [Table 2-4](#) on page 2-14.

**Table 2-4. CRE Initialization Errors**

Error Text	Cause of Error
Backup heap allocation failed	Invalid heap or heap control block.
EXECUTION-LOG has invalid filename ( <i>filename</i> )	Text for PARAM EXECUTION-LOG does not specify a valid external file name.
Invalid environment	The CRE cannot locate the Master Control Block (MCB), or the version level of the MCB is newer than the release level of the CRE.
Invalid environment - Unable to obtain minimum process heap space	CRE could not obtain adequate process heap space.
Invalid startup message volume value ( <i>text</i> )	Invalid volume/subvolume field value; message <i>text</i> appears when the field value is printable.
Log filename unknown	CRE cannot determine the name of the standard log file.
Missing startup message ( <i>nnnnn</i> )	\$RECEIVE read failed, or the wrong message was sent. File-system error <i>nnnnn</i> appears when available.
Not closed by ancestor ( <i>nnnnn</i> )	\$RECEIVE read failed. File-system error <i>nnnnn</i> appears when relevant.
Not opened by ancestor ( <i>nnnnn</i> )	FILE_OPEN_ failed. File-system error <i>nnnnn</i> appears when relevant.
Premature takeover	Backup process took over before process memory was fully initialized.
Signal during initialization	Signal or trap occurred before standard log file initialization was complete.
STDERR assigned invalid filename ( <i>text</i> )	Text in Assign message for STDERR does not specify a valid external file name. Message <i>text</i> appears when available; otherwise, file-system error <i>nnnnn</i> appears if relevant.
Unable to initialize signal handling	Native CRE could not initialize default signal handler.
Unable to open \$RECEIVE ( <i>nnnnn</i> )	FILE_OPEN_ failed. File-system error <i>nnnnn</i> appears when relevant.

If a process receives an open message before process initialization is complete, the CRE returns file-system error 100 (FENOTREADY). In this rare situation, the application returning error 100 must retry the open until the process has received the close message from its creator process.



The following errors can occur during CRE initialization after the CRE has set up its environment. [Section 10, Run-Time Diagnostic Messages](#), describes these errors in greater detail. The CRE writes a message to standard log and terminates your program if any of these errors occur:

Message Number	Cause
17	The CRE could not obtain space from the run-time heap (user data segment).
21	The operating system returned an error when the CRE tried to read an initialization message such as an ASSIGN message or the PARAM message. The message text includes the error number returned by the operating system.
23	The CRE was unable to convert a file name from internal to external format.
24	Multiple ASSIGNS apply to the same file.
25	An ASSIGN of the form <i>filename</i> appears in more than one program.
26	A PARAM does not contain a valid value.
27	A PARAM of the form <i>filename</i> appears in more than one program.
55	A parameter to the CRE initializer is missing or invalid.

## Initializing the TNS CRE From TAL

If your program's main routine is written in TAL and you want to use the services of the CRE, you must call the `TAL_CRE_INITIALIZER_` library procedure when your program begins execution. The `TAL_CRE_INITIALIZER_` library procedure is located in the TAL run-time library, TALLIB. The *TAL Programmer's Guide* describes the `TAL_CRE_INITIALIZER_` library procedure.

## Initializing the Native CRE From pTAL

A pTAL program cannot initialize the native CRE. If your program's main routine is written in pTAL and you want to use the services of the CRE, you must recode the main routine as a normal routine. Then write a C main function that calls the pTAL routine.

## Program Termination

In the Guardian environment, the CRE performs the tasks described in this subsection. In the OSS environment, the CRE does not:

- Ensure that all CRE buffers are empty
- Close standard files

In the TNS environment, if your program uses high-level language constructs to stop its run, such as the COBOL STOP RUN statement, the FORTRAN STOP statement, or

the `C exit()` function, the compiler generates a call to a language-specific run-time library function for program termination. For programs that run in the TNS CRE, the language-specific run-time library function calls the `CRE_Terminator_` procedure.

## CRE\_Terminator\_Procedure

The syntax used to call the `CRE_Terminator_` procedure is described in [Section 6, CRE Service Functions](#). `CRE_Terminator_` performs the following tasks:

- In the TNS environment, it invokes a language-specific termination function for each language, except TAL, represented in your program. In the native environment, it invokes a NonStop operating system routine that calls all shared run-time library termination routines.
- If your program terminates normally or because of a logic problem:
  - Closes standard input if it is open.
  - Ensures that all records in CRE buffers for standard output and standard log are written, and closes standard output and standard log.

The CRE closes only the three standard files. All other files are closed from the run-time library function or user routine that opened them. Otherwise, they are closed implicitly when the CRE calls the `PROCESS_STOP_` system procedure.

- Invokes `PROCESS_STOP_`, passing it `CRE_Terminator_` parameters that correspond to `PROCESS_STOP_` parameters.

## Handling Error Conditions in CRE\_Terminator\_

If an error occurs while `CRE_Terminator_` is executing, `CRE_Terminator_` terminates your program as smoothly as possible by taking the following actions:

- If `CRE_Terminator_` detects that its data has been corrupted, it repairs the corrupted data if possible. If it cannot repair the corrupted data, `CRE_Terminator_` invokes `PROCESS_STOP_`, specifying ABEND and the text “Corrupted environment.”
- If a trap occurs in the TNS environment while `CRE_Terminator_` is terminating your program, `CRE_Terminator_` invokes `PROCESS_STOP_`, specifying ABEND and the text “Trap during termination.”
- If a signal occurs in the native environment while `CRE_Terminator_` is terminating your program, `CRE_Terminator_` invokes `PROCESS_STOP_`, specifying ABEND and the text “Signal during termination.”
- If your program does not pass the `Completion_Status` parameter to `CRE_Terminator_`, `CRE_Terminator_` attempts to write the message “Missing or invalid parameter” to standard log. If for any reason `CRE_Terminator_` cannot write the message, it uses a generic CRE completion error value for the status code and reports the text “Missing or invalid parameter.”

- If `CRE_Terminator_` cannot close a standard file, it writes an error to standard log. If it cannot write to standard log, `CRE_Terminator_` uses a generic completion error value for the status code and reports the text “Close error on IN”, “Close error on OUT”, or “Close error on log” according to which standard file it cannot close.
- If one of the values supplied as a parameter that `CRE_Terminator_` passes on to `PROCESS_STOP_` is rejected, `CRE_Terminator_` invokes `PROCESS_STOP_`, specifying ABEND and the text “Invalid termination parameter.”

When the CRE completes its termination logic, it returns a status code to the process that created your process, typically the TACL command interpreter. The status code specifies the reason your process terminated. See [CRE\\_Terminator](#) on page 6-42 for more details on CRE termination and termination codes.

## Sharing Standard Files

The C, COBOL, and FORTRAN languages provide a predefined input file—called “standard input”—and a predefined output file—called “standard output”. Your program can use standard input and standard output without having to write extensive code to define, open, or manage them. For most languages, a third file—“standard log”—is also predefined. (FORTRAN does not support unit connection to standard log except for messages specified in FORTRAN PAUSE and STOP statements.)

The CRE enables a program to share the standard files—standard input, standard output, and standard log—with routines written in more than one language. With the CRE, shared standard files accessed from routines written in different languages behave the same as if they were unshared and accessed from only one language.

---

**Note.** TNS environment language modules cannot share with native language modules.

---

In the Guardian environment, the CRE performs the tasks described in this subsection. In the OSS environment, the CRE does not support standard files because the OSS file system provides this feature.

## Sharing Standard Files Without Using the CRE

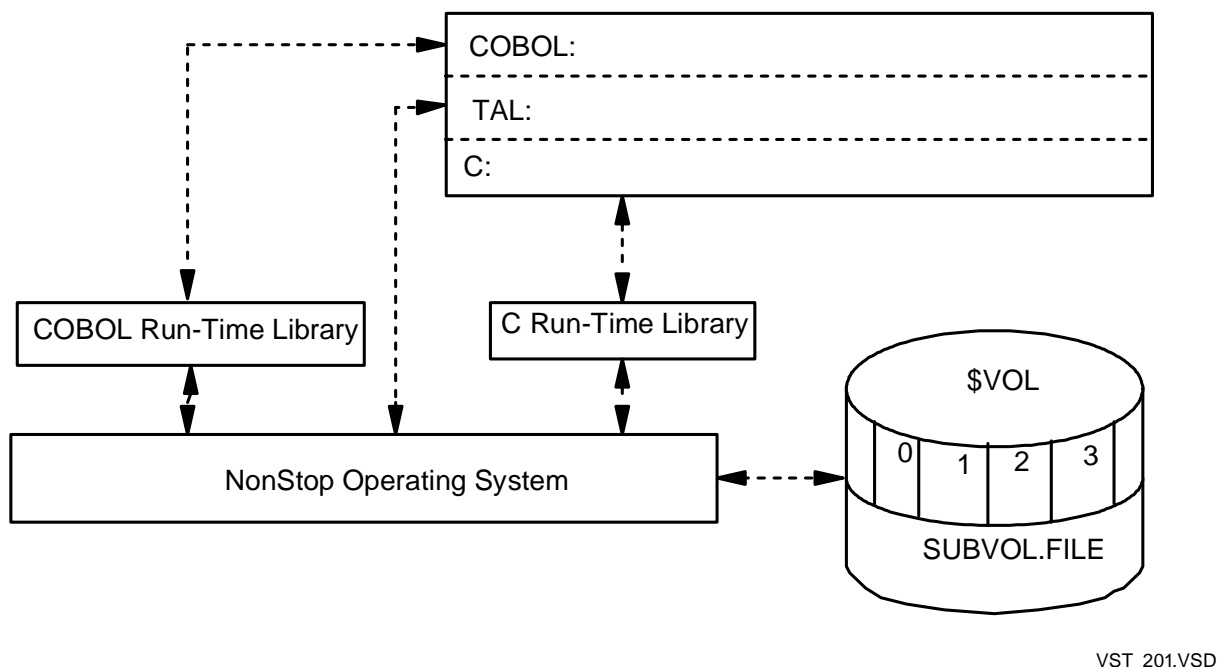
For programs that do not run in the CRE, I/O requests to standard input, standard output, and standard log are processed by the language-specific run-time library for the requesting routine. The run-time library sends requests for system services directly to the HP NonStop operating system. Running mixed-language programs without the CRE is limited because of incompatibilities between run-time libraries.

For example, if you have routines written in C, COBOL, and TAL, requests to write to standard output from C routines are processed by the C run-time library, requests from COBOL routines are processed by the COBOL run-time library, and requests from TAL routines must be sent by the routines directly to the operating system. Because the three routines cannot share the same operating system file open, each establishes a separate operating system open to the file. If routines in each of the three languages open and write to the same disk file, data from each of the three run-time libraries

might write to the same disk location because there is no method for the run-time libraries to share the file record pointer. That is, because each routine begins writing at the beginning of the same file, the first record written by each of the program's routines writes to the same disk location—thereby overwriting data already written there by another routine in the same program.

[Figure 2-1](#) on page 2-18 is an example of a mixed-language TNS program that has COBOL, TAL, and C routines, and does not run in the CRE. The main routine is written in COBOL. The C and COBOL run-time libraries call system procedures to access a file. Because the TAL library does not provide support for input or output, TAL routines must send I/O requests directly to the NonStop operating system.

**Figure 2-1. A C-Series Mixed-Language Process**



## Sharing Standard Files Using the CRE

For Guardian programs that run in the CRE, I/O requests to standard input, standard output, and standard log are processed by the language-specific run-time library for the requesting routine. However, the run-time library requests system services by calling CRE library functions rather than by calling system procedures directly. If the CRE receives a request to open a standard file that it has not opened at the operating system level, it calls the `FILE_OPEN_` system procedure to open the file. Having opened the file, the CRE grants a connection—a path—to the same file open for each additional request that the CRE receives to open the same standard file. The CRE coordinates requests for new connections and requests to release previously granted connections. With the CRE, the standard files belong to a program as a whole, rather than to any one routine or run-time library of a program.

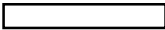

For example, if you have routines written in C, COBOL, and TAL, requests to write to standard output from each of the routines can be coordinated by the CRE. (The C and COBOL run-time libraries call CRE functions directly. Because the TAL run-time library does not support I/O operations, routines written in TAL must call CRE functions directly in order to use the standard files features of the CRE.) If routines in each of the three languages open and write to the same disk file, no data is lost because the CRE enables the routines to share the file record pointer.

Standard input and standard output usually correspond to the files you specify with the IN and OUT parameters when you run a program from TACL. The CRE uses the file names you specify on PARAMs and ASSIGNs to locate standard log. See [Standard Log](#) on page 2-33 for a detailed explanation of how the CRE determines the physical name of standard log.

The CRE opens a standard file only when a run-time library for one of your routines requests that CRE open the file. Standard log, however, is also used by the CRE itself. The CRE might open it before it receives a request from one of your routines. The CRE library, run-time libraries, and your routines can write diagnostic messages to standard log. [CRE File Message](#) on page 6-9 describes how you can write messages to standard log.

The CRE closes a standard file only when all routines with connections to a standard file close the file. Your program might need to close all opens to a file in order to release the file. For example, you might want to print a spooler file without terminating your program.

The following legend applies to [Figure 2-2](#) on page 2-20 through [Figure 2-9](#) on page 2-27.

-----	A dashed line is a path that exists but is not currently active.
—————	A solid line is a path that is active for the current operation.
	A white object is inactive for the current operation.
	A shaded object is active for the current operation.

VST 202.VSD

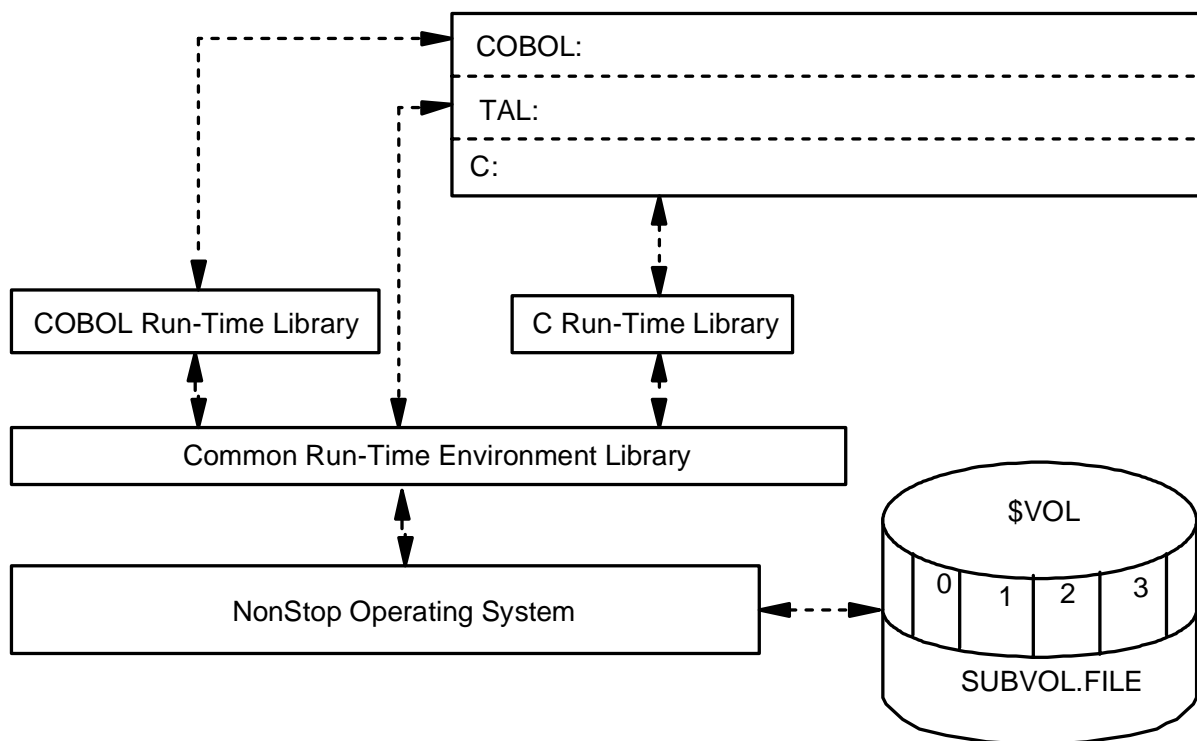
[Figure 2-2](#) through [Figure 2-5](#) show a sequence of opens to standard output by a program consisting of routines written in TNS COBOL, TAL, and C. In the example, standard output is a disk file named \$VOL.SUBVOL.FILE. The main routine is written in COBOL.

- [Figure 2-2](#) on page 2-20 shows a quiescent system. Standard output has not been opened by any routine. There are no operations in progress so all paths are dashed lines and all objects are white boxes.
- [Figure 2-3](#) on page 2-21 shows an active TNS COBOL routine. However, TNS COBOL routines do not explicitly open standard files. Instead, the TNS COBOL run-time library opens standard output the first time a TNS COBOL routine executes a DISPLAY statement.

- [Figure 2-4](#) on page 2-22 shows the TAL routine opening standard output by calling `CRE_File_Open_`. The path from TAL to the CRE is a solid line and, because this is the first time a routine has requested a connection to standard output, the CRE calls the system procedure `FILE_OPEN_`. The path to the operating system is a solid line and the operating system box is shaded. When the call to `FILE_OPEN_` completes, the CRE grants the TAL routine a connection to the file and returns the connection number to the TAL routine.
- [Figure 2-5](#) on page 2-23 shows the C routine opening standard output. Because the main routine is not written in C, the C run-time library does not open standard input, standard output, or standard log when your program begins execution. Therefore, the C routine must call the `fopen_std_file()` library function for each standard file that it accesses.

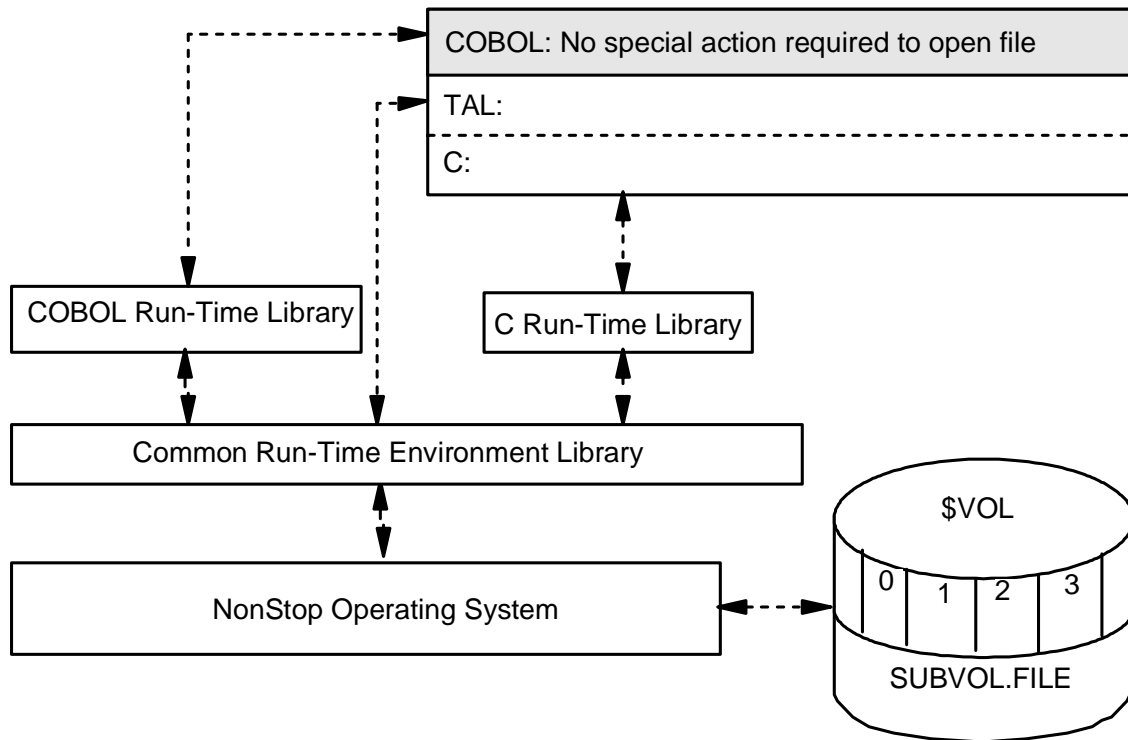
When the C routine calls `fopen_std_file()`, the C run-time library calls the CRE to open standard output. The CRE does not call `FILE_OPEN_` because the CRE already has an open to `$VOL.SUBVOL.FILE` as a result of the preceding TAL request. The CRE grants the C routine a connection to `$VOL.SUBVOL.FILE` and returns the connection to the C run-time library. The operating system box in [Figure 2-5](#) on page 2-23 is not shaded because the CRE does not send an open request to the operating system.

**Figure 2-2. Using the CRE—Mixed-Language Process—Quiescent State**

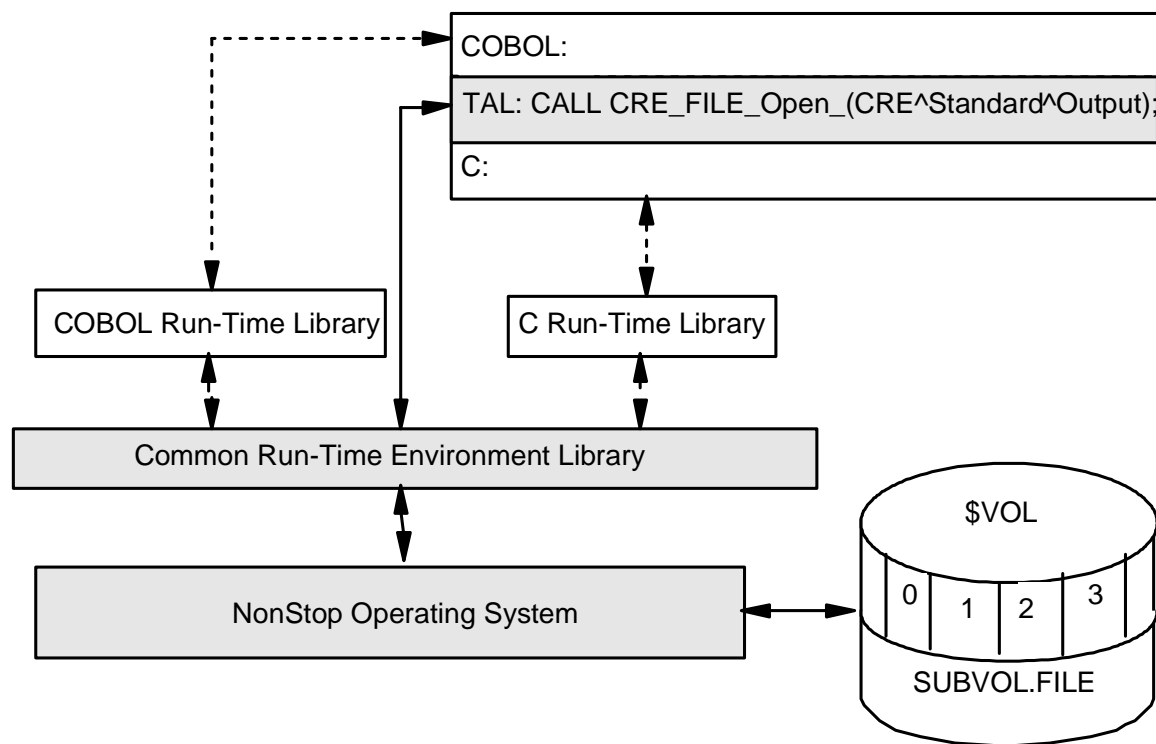


VST 203.VSD

**Figure 2-3. Using the CRE—The COBOL Routine Defaults Opening Standard Output**

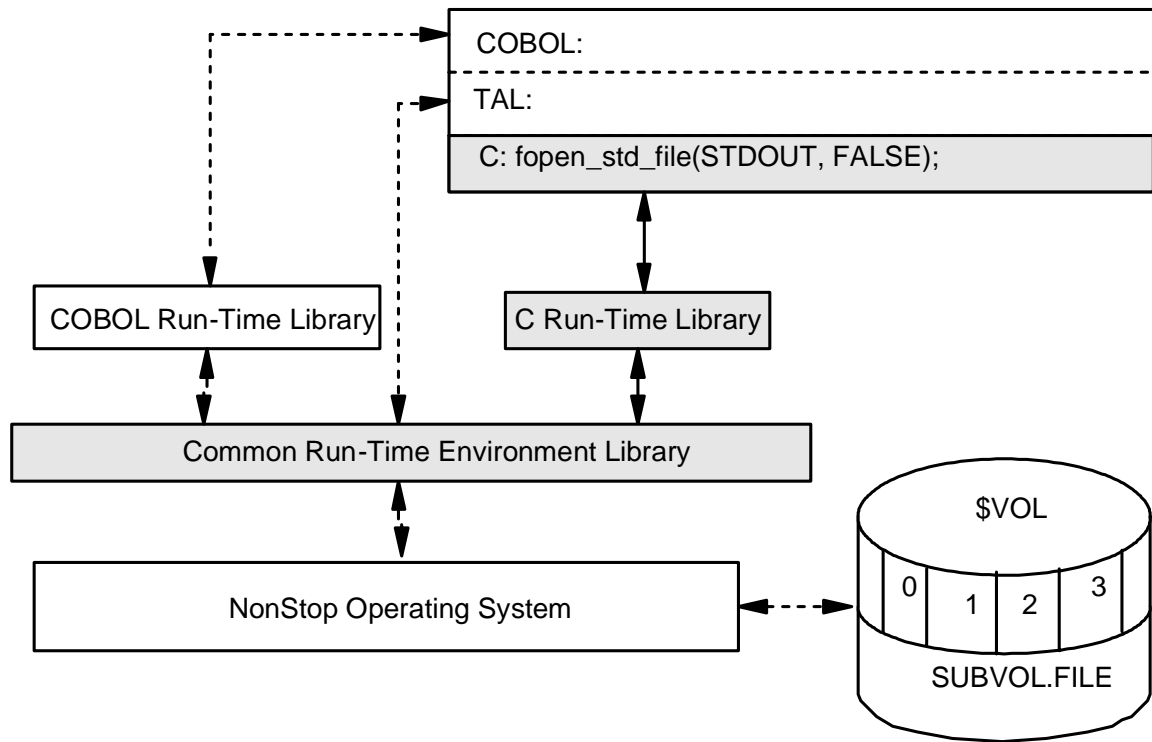


VST 204.VSD

**Figure 2-4. Using the CRE—The TAL Routine Opens Standard Output**

VST 205 .VSD

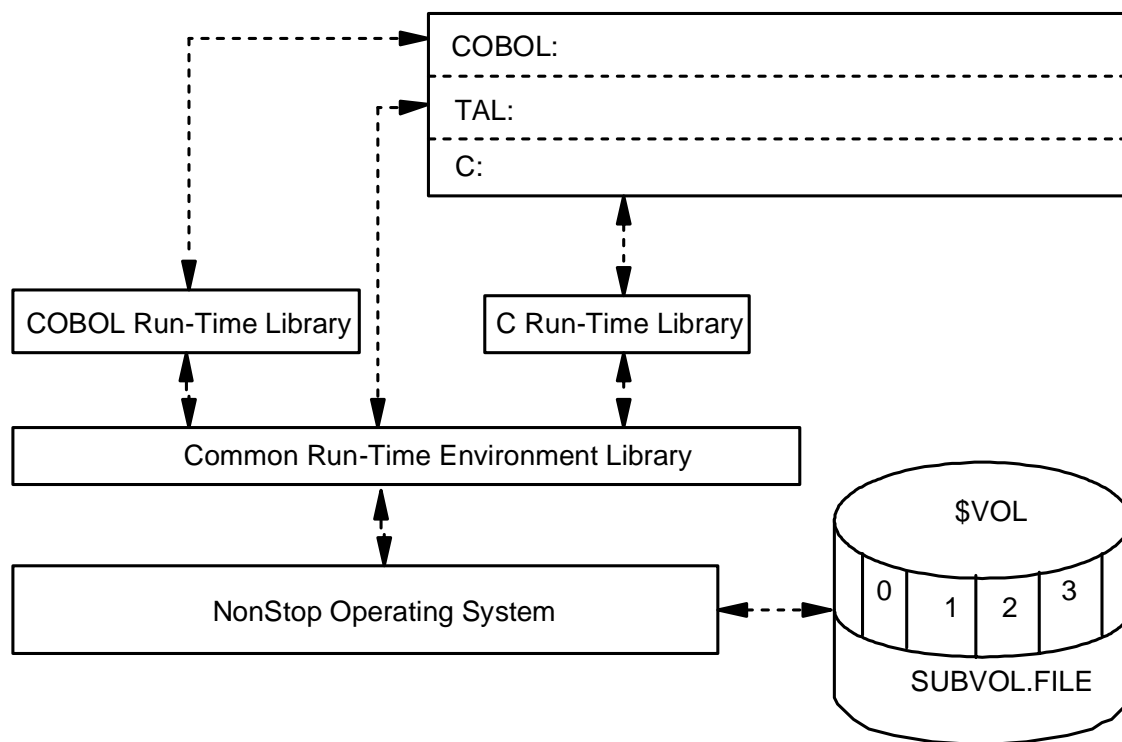


**Figure 2-5. Using the CRE—The C Routine Opens Standard Output**

VST 206.VSD

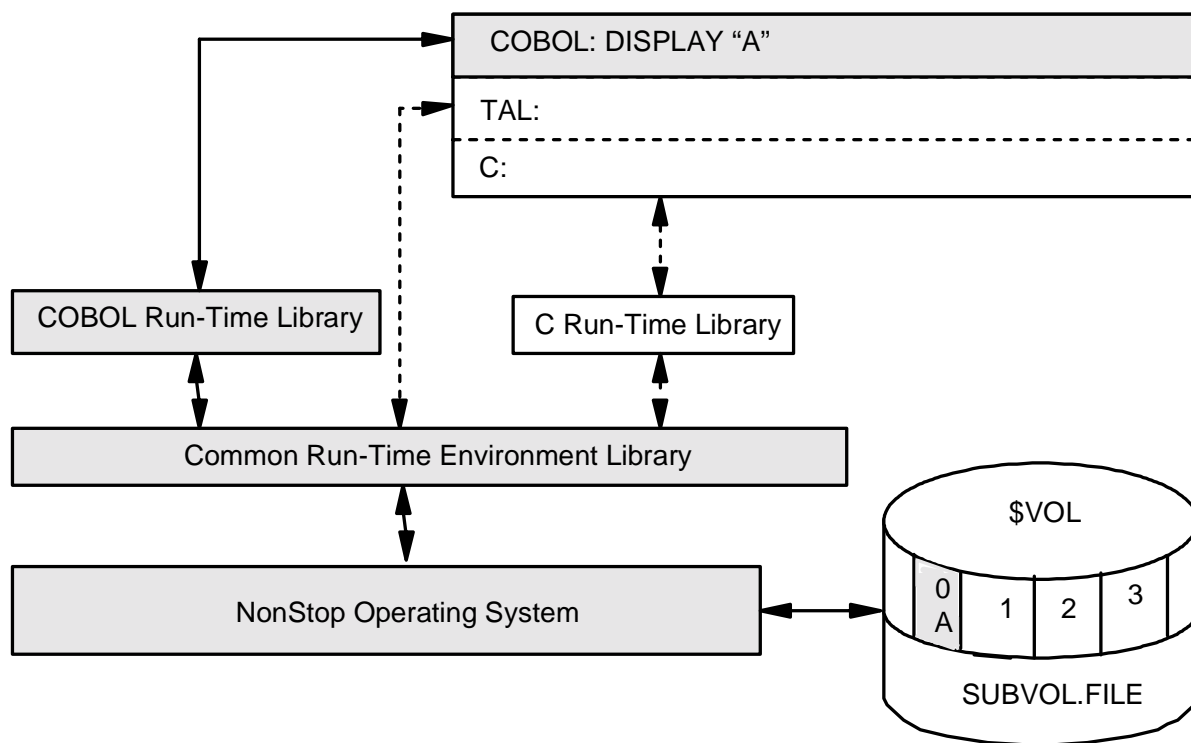
[Figure 2-6](#) through [Figure 2-9](#) repeat the same sequence as shown in [Figure 2-2](#) through [Figure 2-5](#). In [Figure 2-7](#) through [Figure 2-9](#), however, a routine from each of COBOL, TAL, and C writes a record to standard output.

- [Figure 2-6](#) on page 2-24 shows a quiescent system. Standard output has been opened by routines written in C and TAL.
- [Figure 2-7](#) on page 2-25 shows the COBOL routine executing a DISPLAY verb to write the letter A to standard output. Before calling the CRE to write the letter A, the COBOL run-time library calls the CRE to open standard output. The CRE grants a connection to the already open file and returns to the COBOL run-time library, which writes the letter A to standard output. The letter A appears at the first logical location in the file \$VOL.SUBVOL.FILE.
- [Figure 2-8](#) on page 2-26 shows a TAL routine writing a record to standard output. Although this is the first TAL write to standard output, its record, containing the letter B, is written to the second record of standard output.
- [Figure 2-9](#) on page 2-27 shows a C routine writing a record to standard output. Although this is the first C write to standard output, its record, containing the letter C, is written to the third record of standard output.

**Figure 2-6. Using the CRE—Quiescent State With Standard Output Open**

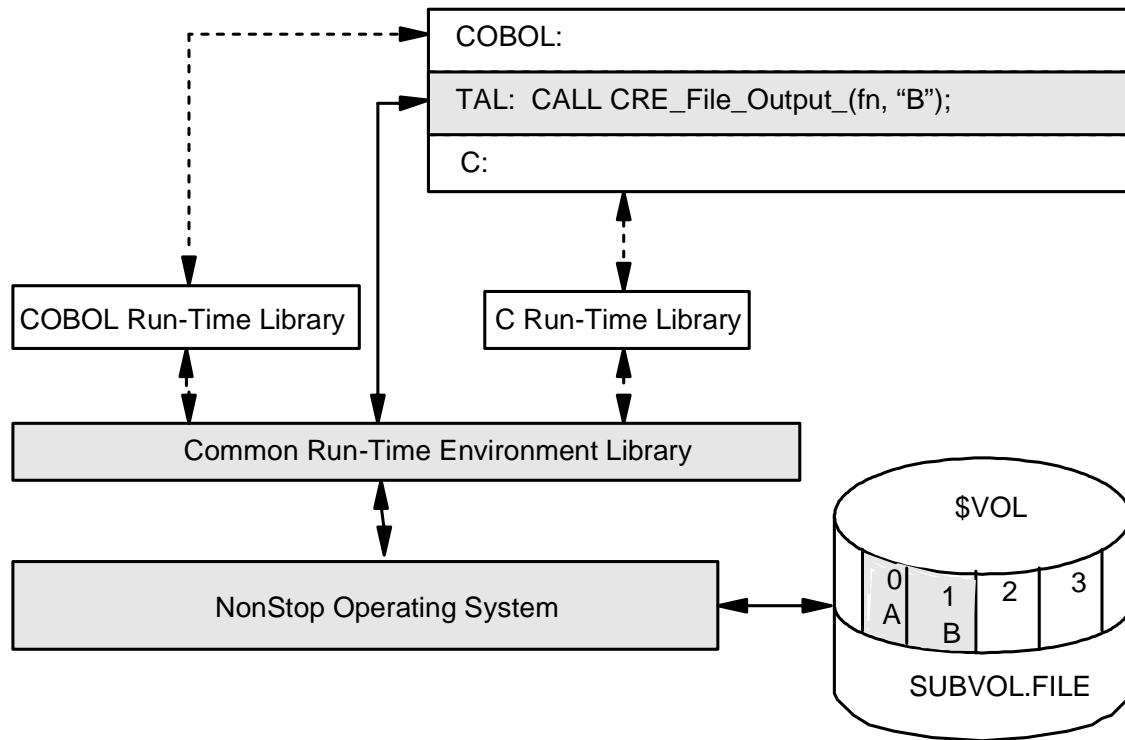
VST 207.VSD

**Figure 2-7. Using the CRE—The COBOL Routine Writes to the File  
\$VOL.SUBVOL.FILE**

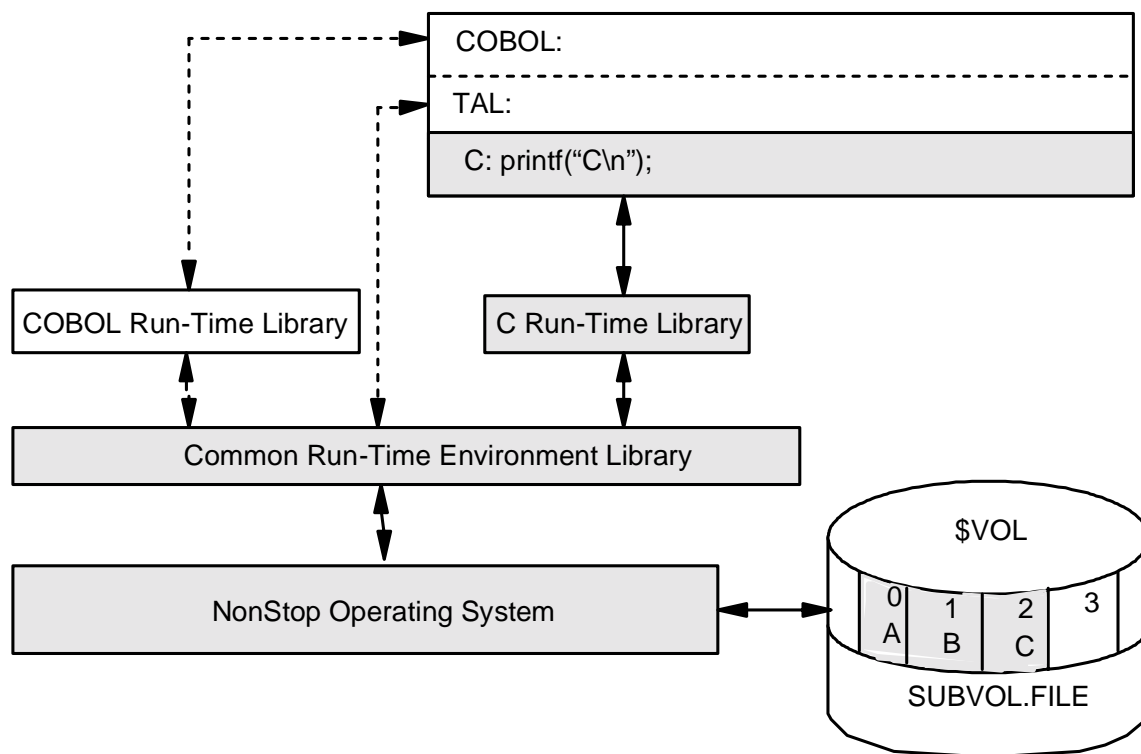


VST 208.VSD

**Figure 2-8. Using the CRE—The TAL Routine Writes to the File \$VOL.SUBVOL.FILE**



VST 209.VSD

**Figure 2-9. Using the CRE—The C Routine Writes to the File \$VOL.SUBVOL.FILE**

VST 210.VSD

## Using CRE Functions to Access the Standard Files

Except under unusual circumstances, use only high-level language constructs to access standard files. For programs running in the CRE, each language-specific run-time library calls CRE functions to process your requests. Under some circumstances, however, you might need to invoke CRE functions or system procedures directly. For example, you might invoke `CRE_File_Control_` to issue a page eject command to a printer. If your program invokes CRE functions or system procedures directly, you must ensure that the routines and procedures you invoke do not alter or interfere with the I/O state maintained by the CRE.

The CRE provides the following functions for standard files. The functions are described in [Section 6, CRE Service Functions](#).

<code>CRE_File_Close_</code>	<code>CRE_File_Open_</code>	<code>CRE_Homterm_Open_</code>
<code>CRE_File_Control_</code>	<code>CRE_File_Output_</code>	<code>CRE_Log_Message_</code>
<code>CRE_File_Input_</code>	<code>CRE_File_Retrycheck_</code>	<code>CRE_Spool_Start_</code>
<code>CRE_File_Message_</code>	<code>CRE_File_Setmode_</code>	

## Determining When Standard Files Are Opened

The language in which the main routine is written determines when and if standard files are opened. If the main routine is written in C, the C run-time library opens the three standard files as part of program initialization unless the C main routine specifies the `nostdfiles` pragma. If the main routine is written in COBOL, FORTRAN, or TAL, standard files are opened only when routines in those languages explicitly execute a statement that opens a standard file.

If a program contains C routines and the main routine is not written in C, one of the C routines must open explicitly each standard file before any C routine in the program accesses that standard file. You open standard files in C by calling the C library function `fopen_std_file()`:

```
fopen_std_file( file, die_on_error )
```

`fopen_std_file()` opens a connection to a standard file.

The *Guardian TNS C Library Calls Reference Manual* describes the `fopen_std_file()` library function.

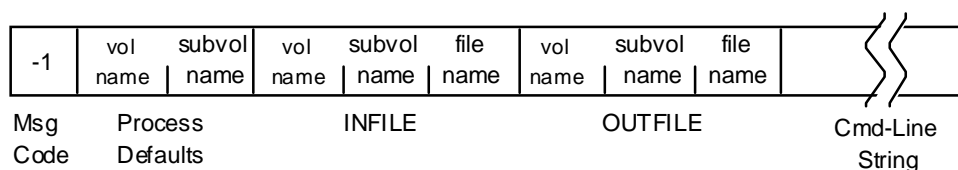
See the reference manual for each language for further information on opening standard files.

## Using Terminals and Process

If the physical file specified for standard input is the same as the physical file specified for either standard output or standard log (or both), and the device type of standard input is either a terminal or a process (but not a spooler collector), the CRE opens the specified file only once and uses the same operating system file open when your program reads from standard input or writes to standard output or standard log.

## Program Startup Message

When your program begins execution, the CRE reads the startup message sent to it by the process that starts your program. The discussions that follow frequently reference two of the fields of the startup message: INFILE and OUTFILE. [Figure 2-10](#) shows the locations of INFILE and OUTFILE in the startup message. The *Guardian Programmer's Guide* describes all of the fields in the startup message.

**Figure 2-10. Process Startup Message Layout**

VST 21 1VSD

## Standard Input

Standard input is a special file that is available to all programs that use the CRE. You must open it for read-only access.

The device for standard input must support sequential read access. The CRE supports the following devices for standard input:

- A process (other than a spooler collector)
- \$RECEIVE, described in [Using \\$RECEIVE](#) on page 2-34
- A disk file
- A terminal

The CRE determines the file name for standard input as follows. If the INFILE name in your program's startup message is:

- Not the name of your program's home terminal, the CRE uses the INFILE name from the startup message for standard input.
- Blanks, the CRE does not open a system file but accepts open requests and returns end of file each time your program reads from standard input.
- The name of your program's home terminal and you do not specify the EXECUTION-LOG PARAM, the CRE opens your home terminal if your program opens standard input.
- The name of your program's home terminal and you specify a file name as the EXECUTION-LOG, the CRE uses the file you specify for the EXECUTION-LOG as standard input.

For example, if your home terminal is named \$TERM and you specify AFILE for EXECUTION-LOG, the CRE opens AFILE if a routine in your program opens standard input:

```
PARAM EXECUTION-LOG AFILE
RUN myprog / IN $TERM, ..... /
```

If you do not specify the IN parameter, TACL passes the name of your terminal as the IN parameter, which has the same effect as explicitly specifying your home terminal as the IN parameter:

```
PARAM EXECUTION-LOG AFILE
RUN myprog
```

- The name of your program's home terminal and you specify an asterisk for the EXECUTION-LOG, the CRE does not open a file for standard input. Instead, the CRE returns end of file each time your program reads from standard input:

```
PARAM EXECUTION-LOG *
RUN myprog / IN $TERM, ..... /
```

If you do not specify the IN parameter, TACL passes the name of your terminal as the IN parameter, which has the same effect as explicitly specifying your home terminal as the IN parameter:

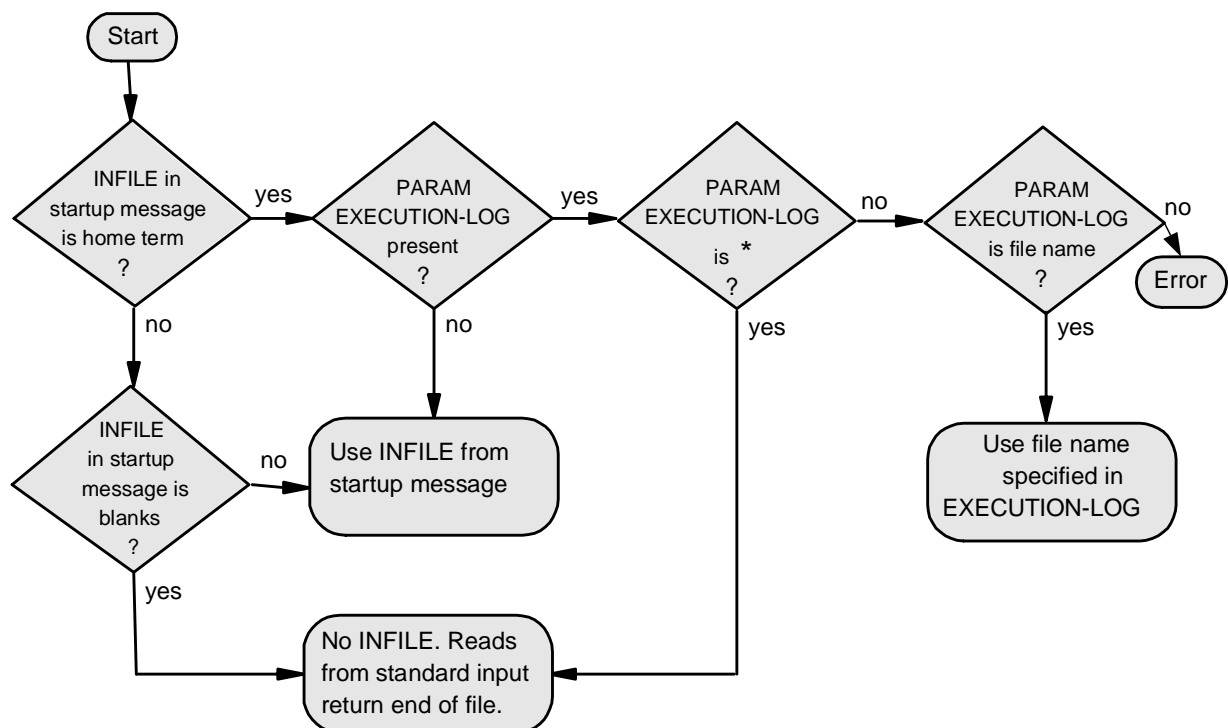
```
PARAM EXECUTION-LOG *
RUN myprog
```

---

**Note.** The CRE does not recognize a TACL ASSIGN for a logical file called STDIN.

---

**Figure 2-11. Establishing the File Name of Standard Input**



VST 212.VSD



If you start your program from a TACL command line or a TACL macro, the INFILE in the startup message is the name you specify in the TACL IN parameter or, if you do not specify an IN parameter, then the name by which TACL knows your terminal.

If a program other than TACL starts your program, INFILE in the startup message contains whatever file name the other program stores when it creates the startup message.

## Standard Output

Standard output is a special file that is available to all programs that use the CRE. You must open it for write-only access.

The device for standard output must support sequential write access. The CRE supports the following devices for standard output:

- A process (including a spooler collector)
- An operator console
- A disk file
- A terminal
- A printer

The CRE determines the file name for standard output as follows. If the OUTFILE name in your program's startup message is:

- Not the name of your program's home terminal, the CRE uses the OUTFILE name from the startup message for standard output.
- Blanks, the CRE does not open a system file but accepts open requests, discards records that you write to standard output, and indicates a successful write each time your program writes to standard output.
- The name of your program's home terminal and you do not specify the EXECUTION-LOG PARAM, the CRE opens your home terminal if your program opens standard output.
- The name of your program's home terminal and you specify a file name as the EXECUTION-LOG, the CRE uses the file you specify for the EXECUTION-LOG as standard output.

For example, if your home terminal is named \$TERM and you specify AFILE for EXECUTION-LOG, the CRE opens AFILE if a routine in your program opens standard output:

```
PARAM EXECUTION-LOG AFILE
RUN myprog / OUT $TERM, ..... /
```

Note that if you specify an EXECUTION-LOG PARAM, the CRE will not write to your home terminal under any circumstances, even if EXECUTION-LOG specifies the name of your home terminal.

- The name of your program's home terminal and you specify an asterisk for the EXECUTION-LOG, the CRE does not open a file for standard output. Instead, each

time your program writes to standard output, the CRE discards the record and indicates a successful write:

```
PARAM EXECUTION-LOG *
RUN myprog / OUT $TERM, ..... /
```

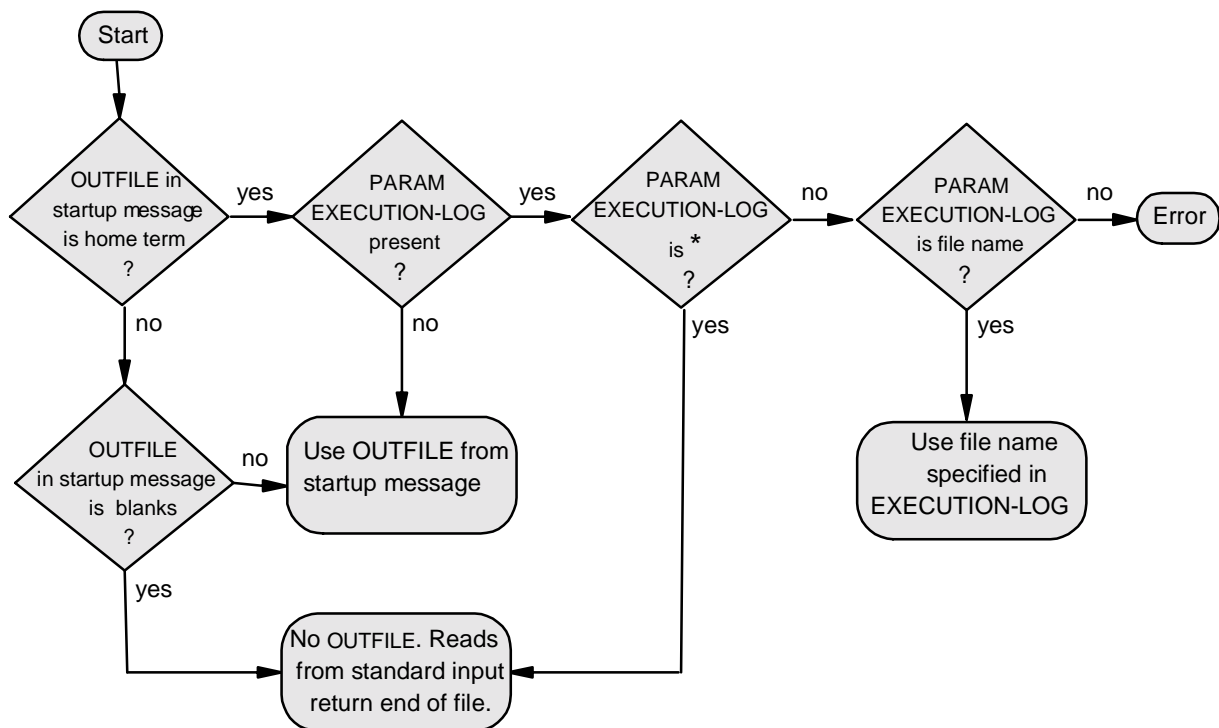
- The name of your program's home terminal and you specify the program's home terminal as the EXECUTION-LOG, the CRE uses the file you specify for the EXECUTION-LOG as standard output.

---

**Note.** The CRE does not recognize a TACL ASSIGN for a logical file called STDOUT.

---

**Figure 2-12. Establishing the File Name of Standard Output**



VST 213VSD

If you start your program from a TACL command line or TACL macro, the OUTFILE in the startup message is the name you specify in the TACL OUT parameter or, if you do not specify an OUT parameter, then the name by which TACL knows your terminal.

If a program other than TACL starts your program, the OUTFILE in the startup message contains whatever file name the other program specifies when it creates the startup message.

## Standard Log

Standard log is a special file that is available to all programs that use the CRE. If you open standard log explicitly, you must open it for write-only access. Your program does not have to declare it or specify its attributes although it can do so. The CRE and run-time libraries write messages to standard log such as error messages, warnings, and informational messages. A user program can also send messages to standard log by calling `CRE_Log_Message_`. See [CRE\\_Log\\_Message\\_](#) on page 6-25.

A FORTRAN program cannot open standard log using a default unit number, as it can standard input (UNIT 5) and standard output (UNIT 6). A FORTRAN program can, however, use the FORTRAN routines `PAUSE` or `STOP` to write a message to standard log. The *FORTRAN Reference Manual* describes the `PAUSE` and `STOP` statements.

The device for standard log must support sequential write access. The CRE supports the following devices for standard log:

- A process (including a spooler collector)
- The operator console
- A disk file
- A terminal
- A printer

By default, the CRE writes log messages to your process's home terminal. You can direct log messages to another file, however, by specifying a file name in a `TACL ASSIGN` command or `PARAM` command.

- If an `ASSIGN` specifies the logical name `STDERR`, the CRE uses the physical name from the `ASSIGN` as the name of standard log.
- If a `PARAM` specifies `EXECUTION-LOG`, the CRE uses the value of the `EXECUTION-LOG PARAM` as the name of standard log. If the `EXECUTION-LOG PARAM` specifies an asterisk (\*), the CRE does not open a file for standard log and discards messages that your program writes to standard log.
- If an `ASSIGN` specifies `STDERR` and a `PARAM` specifies `EXECUTION-LOG`, the CRE uses the physical name from the `ASSIGN` unless the physical name specifies your home terminal, in which case the CRE uses the value of the `EXECUTION-LOG PARAM` or, if the `EXECUTION-LOG PARAM` value is an asterisk, the CRE does not open standard log.

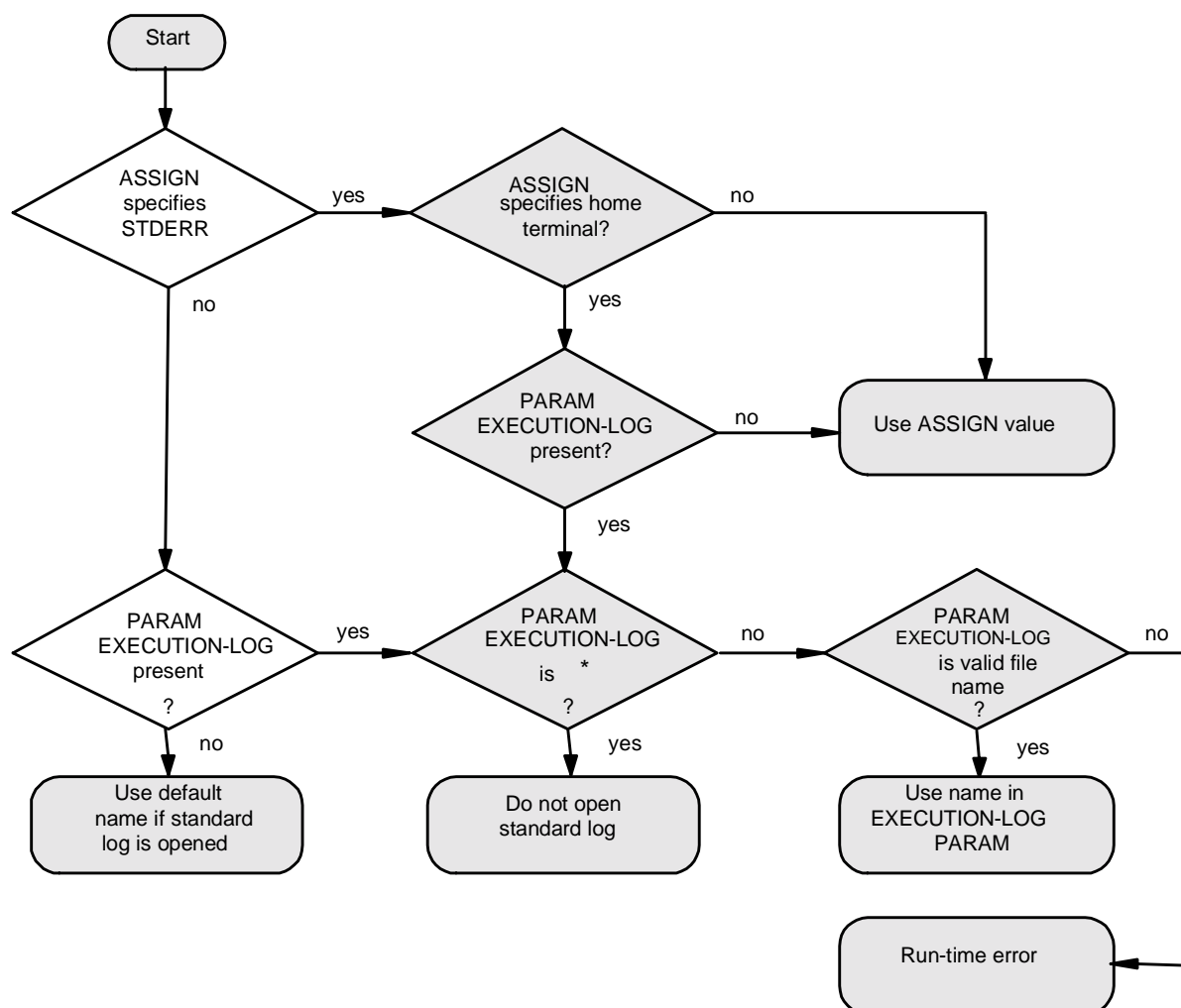
The CRE does not open a file for standard log if the `ASSIGN` for `STDERR` specifies either:

- Blanks for the file name
- The process's home terminal, and the value of the `EXECUTION-LOG PARAM` is an asterisk

Although the CRE might not open a file for standard log, it accepts requests to open standard log and requests to write to standard log, but the CRE discards the data you write and indicates a successful write.

The flow chart in [Figure 2-13](#) on page 2-34 shows how the CRE determines the name of standard log.

**Figure 2-13. Establishing the File Name of Standard Log**



VST214.VSD

## Using \$RECEIVE

In both the TNS and native environments, processes have access to a special file called \$RECEIVE. Your process receives messages sent to your process from other processes (from the operating system, from your backup process, and so forth) by reading \$RECEIVE. Like standard files discussed earlier in this section, \$RECEIVE is a resource that is available equally to all routines in your process, not to any one routine or language-specific run-time library.

In the Guardian environment, the CRE performs the tasks described in this subsection. Processes running in the OSS environment do not receive startup, ASSIGN, or

PARAM messages; the CRE, therefore, does not process such messages when it runs in the OSS environment.

## **\$RECEIVE and Program Initialization**

During program initialization, the CRE reads initialization messages from \$RECEIVE. At the end of program initialization, the CRE closes \$RECEIVE. \$RECEIVE remains closed until the CRE receives a request to open it or the CRE itself needs to open it.

## **Messages Received From \$RECEIVE**

Your program can receive the following kinds of messages from \$RECEIVE:

- System messages that alert your program to important system state changes (for example, “CPU down” messages, process ABEND messages, and so forth).
- System messages sent by the operating system because a process has called a system procedure that addresses your process (for example, calls to FILE\_OPEN\_, SETMODE, CONTROL, and so forth).
- Messages sent to your process by other processes.

## **\$RECEIVE and the Languages Supported by the CRE**

You can read \$RECEIVE from any language supported by the CRE. There are two ways to access \$RECEIVE, by calling either CRE functions only or system procedures only.

### **TNS CRE**

A program can use TNS CRE functions to access \$RECEIVE.

### **Native CRE**

A program that runs in the native CRE can use native CRE functions to access \$RECEIVE.

## **Using CRE Functions to Access \$RECEIVE**

When a COBOL or FORTRAN routine opens \$RECEIVE, the language-specific run-time library calls the CRE functions that manage \$RECEIVE. C or C++, and TAL or pTAL routines can participate in this model by calling the CRE functions that manage \$RECEIVE. The CRE grants a connection for each open, but all connections share the same file attributes and the same operating system file open. Functions that manage \$RECEIVE are described in [Section 6, CRE Service Functions](#).

The CRE opens \$RECEIVE for exclusive access. If the CRE opens \$RECEIVE, no other routine in your program can open it. If a routine in your program opens \$RECEIVE by a direct call to FILE\_OPEN\_, or if a C run-time library opens

\$RECEIVE, the CRE will not be able to open \$RECEIVE because it requires exclusive access.

To access \$RECEIVE, a COBOL or FORTRAN run-time library function, or C/C++, TAL, or pTAL routine opens \$RECEIVE by calling `CRE_Receive_Open_Close_` with the open variant. The run-time library function or other-language routine reads messages from \$RECEIVE by calling `CRE_Receive_Read_` and replies to messages from \$RECEIVE by calling `CRE_Receive_Write_`.

## Using System Procedures to Access \$RECEIVE

Routines written in any of the languages supported by the CRE can access \$RECEIVE by calling system procedures directly. All HP languages include syntax to call system procedures. To manage \$RECEIVE, routines open it by calling the `FILE_OPEN_` system procedure. Routines read and reply to messages from \$RECEIVE using the `READUPDATEX` and `REPLYX` system procedures. If you access \$RECEIVE using system procedures, do not use language features that access \$RECEIVE for you.

## Using a Spooler Collector

You can specify a spooler collector as the device for either standard output or standard log. The following considerations apply to using spooling with the CRE:

- The CRE uses buffered (level-3) spooling for standard output unless you specify unbuffered spooling.
- You cannot use buffered spooling for standard log. (If you did, you might lose messages if your program terminates abnormally—precisely the case for which you want to see the last messages written to standard log.)
- `CRE_File_Open_` does not invoke `CRE_Spool_Start_` or the `SPOOLSTART` system procedure. However, `CRE_File_Output_` invokes `CRE_Spool_Start_` if `CRE_File_Open_` opens a file that is a spooler collector.
- A TAL routine only needs to call `CRE_Spool_Start_` explicitly to specify parameters to `CRE_Spool_Start_` that are not the default parameters. For example, `CRE_Spool_Start_` writes one copy of the specified file to the spooler collector. If your program requires more than one copy of the output, call `CRE_Spool_Start_` after you call `CRE_File_Open_` and specify the *number\_of\_copies* parameter.

In the Guardian environment, the CRE performs the tasks described in this subsection. In the OSS environment, the CRE does not support any of the tasks described in this subsection because the CRE provides spooling features only to support standard files and it does not support standard files in the OSS environment.

For more details, see [CRE\\_Spool\\_Start\\_](#) on page 6-27.

# Memory Organization

This subsection describes the overall layout of memory for programs that use the services of the CRE, and the data spaces that your program can use to share data.

## TNS CRE Memory

In the TNS CRE, the data spaces that your program can use to share data are:

- The extended stack
- The user heap
- Global data

The CRE allocates memory using the same layout in the OSS and Guardian environments. For a given total amount of memory allocated to a process, however, the amount of memory available in the OSS environment might be somewhat different than the amount available to the same process running in the Guardian environment.

## Overall Memory Organization

The TNS CRE, the language-specific run-time libraries, and your code share the same address space for their data, although each has its own block of memory within that space. The TNS CRE and run-time libraries allocate their data in the upper 32K words of the user data segment and in extended memory. If your program specifies a small-memory model, the TNS CRE allocates the heap in the lower 32K words of your user data segment, just below the run-time stack. If your program specifies a large-memory model or a wide-memory model, the TNS CRE allocates the heap as the last data block in the extended segment.

If you use the TNS CRE, Binder ensures that the data structures for the language-specific run-time libraries and for the TNS CRE do not overlay each other.

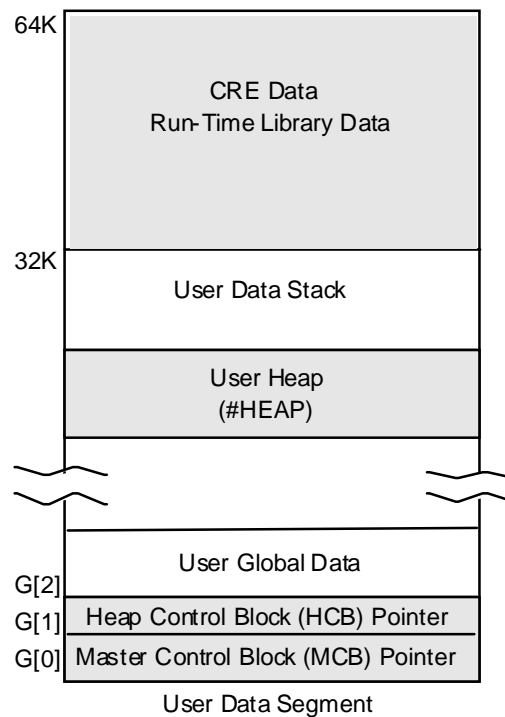
Your program maintains its data in the lower 32K words of the user data segment and in extended memory.

[Figure 2-14](#) on page 2-38 shows the overall memory organization for a program that runs in a small-memory model and uses the TNS CRE. [Figure 2-15](#) on page 2-39 shows the overall memory organization for a program that runs in a large-memory model or a wide memory model, and that uses the TNS CRE. Shaded boxes in the figures contain TNS CRE or language-specific run-time library data. Note that:

- The TNS CRE and run-time libraries store their data in the upper 32K words of the user data segment, beginning at the 32K-word boundary.
- The TNS CRE allocates the user heap in the lower 32K words of the user data segment just below the data stack in a small-memory model ([Figure 2-14](#) on page 2-38). The TNS CRE allocates the user heap in the extended memory segment in a large-memory model and a wide-memory model ([Figure 2-15](#) on page 2-39).

- TNS CRE data in the upper 32K words includes pointers to additional data items, some of which might be in extended memory.
- The TNS CRE maintains a pointer at word 0 of the user data segment (G[0]) to the TNS CRE's Master Control Block (MCB). The TNS CRE and the run-time libraries use the MCB pointer at G[0] to locate their private data.
- The TNS CRE maintains a pointer at word 1 of the user data segment (G[1]) to the TNS CRE's Heap Control Block (HCB).

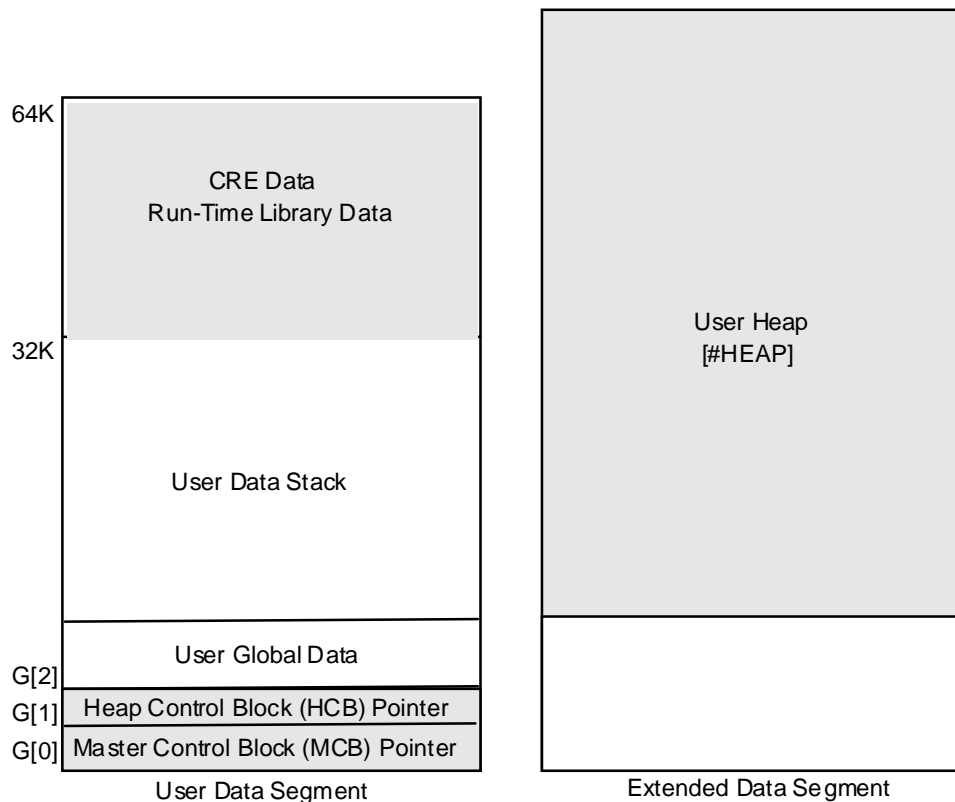
**Figure 2-14. Organization of a Small-Memory-Model Program Running in the TNS CRE**



VST 215VSD



**Figure 2-15. Organization of a Large-Memory- or Wide-Memory Model Program Running in the TNS CRE**



VST 216VSD

## The Extended Stack

The TNS CRE works with the extended stack model used by both TAL and FORTRAN. No other languages use this model.

The model of the extended stack uses three data blocks:

- `$EXTENDED#STACK` is the extended stack itself, allocated in extended memory.
- `EXTENDED#STACK#POINTERS` is a structure with two fields.
  - `#SX` acts as the extended stack S-register.
  - `#MX` points to the last usable byte of memory.
- `EXTENDED#STACK#FRAME` is a single field L-register.

## The User Heap

The TNS CRE manages the heap and allocates blocks from the heap to any routine in your program that requests heap space. The C language includes syntax to request

heap space. TAL routines cannot access the TNS CRE functions that allocate heap space. Data allocated on the heap by a routine written in one language can be accessed by a routine written in another language by passing the address of the data from one routine to another. Routines that access the same data on the heap must specify the same data layout, even if you must use language-specific syntax in each language to describe the same data.

The heap is maintained in a data block called #HEAP. The TNS CRE allocates space for #HEAP in the user data segment (just below your program's run-time stack) if your program uses a small-memory model, or in extended memory (after all other data blocks) if your program uses a large-memory model or a wide-memory model.

## Specifying the Size of the Heap

You can specify the size of the heap using:

Construct	Product	Reference
HEAP pragma	C	<i>C/C++ Programmer's Guide</i>
HEAP compiler directive	TAL	<i>TAL Reference Manual</i>
SET HEAP command	Binder	<i>Binder Manual</i>

---

**Note.** The TNS CRE does not support the HEAP PARAM. In the Guardian environment, you can use compiler directives or Binder commands to achieve the same effect as the HEAP PARAM.

---

The TNS CRE uses the heap size you specify as the initial size of the heap.

If a program uses a small-memory model, the heap is allocated in the user data segment, just below the data stack. The TNS CRE uses the size you specify as the initial size of the heap, but the TNS CRE does not increase the size of the heap once your program begins running.

If a program uses a large-memory model or a wide-memory model, the heap is the last block in the extended memory segment. The TNS CRE increases the size of the heap, if needed, up to the maximum size of an extended memory segment, 127.5 MB. Your program requires disk space to accommodate the extended memory segment. Therefore, set the initial size of the heap to be only as large as your program requires. Use the heap statistics feature to help you determine the size of the heap.

Requesting Space on the Heap:

- Only C routines can request space on the heap. Neither COBOL nor FORTRAN supports pointers and, therefore, cannot request or access heap data. TAL supports pointers but cannot call the TNS CRE to request heap space. A TAL routine can, however, call a C routine to obtain heap space, after which the TAL routine can access heap data using the pointer returned to it by the C routine.
- C routines request heap space by calling the C `malloc()` library function, and return heap space by calling the C `free()` library function.

### Accessing Heap Data:

- C and TAL routines can reference data allocated on the heap by storing the address of the data into a pointer and referencing the pointer in an expression. Routines written in COBOL and FORTRAN must call C or TAL routines to reference heap data.
- A TAL routine can call a C routine to allocate or free heap space. The following code fragment shows a TAL routine, `Do_It`, that calls a C routine, `GETSPACE`, to get space from the heap. `GETSPACE` gets a block from the heap and stores a value in the first 16-bit word of the block. When `GETSPACE` returns, its value is a pointer to the newly allocated block. `Do_It` reads the value stored by `GETSPACE`. The C routine is shown first:

```
#include <stddef.h> nolist
#include <stdlib.h> nolist
int *GETSPACE( int nbytes )
{
    int *p;
    p = (int *) malloc( nbytes )
    if (p != NULL) *p = 100;
    return p;
}
```

The following TAL code calls the C routine `GETSPACE`:

```
INT(32) PROC TAL_Malloc = "GETSPACE" ( nbytes ) LANGUAGE C;
    INT nbytes;
EXTERNAL;

PROC Do_It;
BEGIN
    INT .EXT T_Ptr;

    @T_Ptr := TAL_Malloc( 1000 );
    IF (@T_Ptr = 0D) OR (T_Ptr <> 100) THEN
    BEGIN
        ! Handle error...
    END;
END;
```

### Heap Statistics:

- The TNS CRE maintains statistics that describe heap utilization.
- The native CRE does not support statistics that describe heap utilization.

## Native CRE Memory

In the native CRE, the data spaces that your program can use to share data are:

- The user heap
- Global data

The native CRE allocates memory using the same layout in the OSS and Guardian environments. For a given total amount of memory allocated to a process, however,

the amount of memory available in the OSS environment might be somewhat different than the amount available to the same process running in the Guardian environment.

## Overall Memory Organization

The following table shows the maximum size for the memory segments in the native run-time environment.

Memory Segment	Maximum Size
Total of User Heap and Global Variables	128 MB for pre-G05 systems
Total of User Heap, Global Variables, and Flat Segments	1120 MB for G05 and later systems
Stack	32 MB

## The User Heap

The native CRE manages the heap and allocates blocks from the heap to any routine in your program that requests heap space. The C and C++ languages include syntax to request heap space. pTAL routines can access the native CRE functions that allocate heap space. Data allocated on the heap by a routine written in one language can be accessed by a routine written in another language by passing the address of the data from one routine to another. Routines that access the same data on the heap must specify the same data layout, even if you must use language-specific syntax in each language to describe the same data.

The native CRE selects an appropriate initial size for the user heap. The native CRE can increase the size of the user heap once your program begins running if more heap space is needed.

## Specifying the Size of the Heap and the Stack

You can specify a smaller maximum size for the heap using a linker utility.

The main stack has a default limit of 1 MB on G-series systems and 2 MB on H-series systems, but you can increase this to a maximum of 32 megabytes either by calling the `PROCESS_LAUNCH_` procedure or by using a linker utility.

See the:

- *nld Manual*
- *noft Manual*
- *ld Manual*
- *eld Manual*
- *Guardian Programmer's Guide*

for more information.

## Requesting Space on the Heap

C routines request heap space by calling the C `malloc()` function, and return heap space by calling the C `free()` function. pTAL routines do the same. C++ routines request heap space by calling the `new()` function, and return heap space by calling the `delete()` function.

Accessing Heap Data:

- C, C++, pTAL routines can reference data allocated on the heap by storing the address of the data into a pointer and referencing the pointer in an expression.
- A pTAL routine can call a C or C++ routine to allocate or free heap space. The following code fragment shows a pTAL routine, `Do_It`, that calls a C routine, `GETSPACE`, to get space from the heap. `GETSPACE` gets a block from the heap and stores a value in the first 16-bit word of the block. When `GETSPACE` returns, its value is a pointer to the newly allocated block. `Do_It` reads the value stored by `GETSPACE`. The C routine is shown first:

```
#include <stddef.h> nolist
#include <stdlib.h> nolist
int *GETSPACE( int nbytes )
{
    int *p;
    p = (int *) malloc( nbytes )
    if (p != NULL) *p = 100;
    return p;
}
```

The following pTAL code calls the C routine `GETSPACE`:

```
INT(32) PROC TAL_Malloc = "GETSPACE" ( nbytes ) LANGUAGE C;
    INT(32) nbytes;
EXTERNAL;

PROC Do_It;
BEGIN
    INT .EXT T_Ptr;

    @T_Ptr := TAL_Malloc( 1000D );
    IF (@T_Ptr = OD) OR (T_Ptr <> 100) THEN
    BEGIN
        ! Handle error...
    END;
END;
```

## Using the Native Heap Managers

Beginning at the G06.15 release, the native CRE includes two heap managers:

- The high-performance heap manager, contained in product T1269G09 (NSK CRE/RTL).

The new heap manager is available only for programs that use the native CRE; this includes native C, native C++, and ECOBOL or NMCOBOL programs.

- The original heap manager, contained in product T8431G09 (Native CRE/RTL). This heap manager is not available on H-series systems.

Both native heap managers offer two external features:

- The overwrite released-space feature, which you can use at run time to detect errors in handling heap memory in your applications
- Programmatic setting of heap-management attributes, which you can use to monitor particular attributes of heap management

## Undetected Logic Errors Can Exist in Code that Uses the Original Heap Manager

When a program releases dynamically allocated (heap) memory, the content of the freed memory block is no longer valid and should not be accessed as if it were still allocated.

Space can be released to the heap manager in several ways, including explicit calls to the `free()` function or the C++ `delete` function. In addition, destruction of a C++ object can involve implicit release operations, and programs that use packages such as `tools.h++` typically create instances of implicit release operations when they delete members of collections or elements of lists managed by these packages.

With the new high-performance heap manager, data in freed blocks is more likely to be overwritten than with the original heap manager. For this reason, in some cases client programs using the original heap manager could refer to data values in freed blocks with impunity. However, if a client program using the new heap manager refers to data values in freed memory blocks, unexpected results may occur, and the program may fail altogether.

Therefore, HP recommends that you verify that your applications do not attempt to access data in space that has been released. You can verify your applications by using the overwrite released space feature.

## Using the Overwrite Feature to Detect Logic Errors

If your NonStop server is running the NSK CRE/RTL (T1269), all your processes use the new high-performance heap manager and they must be correct in their heap usage.

You can identify programs that perform erroneous memory handling by using the overwrite released-space feature. You can enable this feature by setting `DEFINES` or `PARAMS`, or by programmatically setting the `RTL^Heap^erase^on^free` attribute, described in [Table 2-5](#).

When the overwrite feature is enabled, if a program retrieves data from space it has released (an invalid action), the program will no longer obtain the values that resided in that space before the space was released. The value obtained may cause the program to behave differently than intended, thus alerting you that invalid logic exists in the program.

## Guardian Commands to Enable Overwrite

To enable the overwrite released-space feature, use either the ADD DEFINE or the PARAM command:

- `ADD DEFINE =_ERASE_ON_FREE_, CLASS MAP, FILE ON`

This DEFINE activates the overwrite released-space feature for each process subsequently initiated by that TACL session. Note that this DEFINE influences processes and is propagated or suppressed according to the rules for all DEFINES.

- `PARAM -ERASE-ON-FREE- {ON|OFF}`

The PARAM -ERASE-ON-FREE- can enable or disable the overwrite released-space feature, and the PARAM takes precedence over the DEFINE =\_ERASE\_ON\_FREE\_ when both are active. Thus, you can disable the overwrite feature using the PARAM command even when the DEFINE =\_ERASE\_ON\_FREE\_ is active.

## OSS Commands to Enable Overwrite

Use either the ADD DEFINE or the EXPORT shell directive as follows:

- `add_define =_ERASE_ON_FREE_ class=MAP file=ON`

This DEFINE activates the overwrite released space feature for each process subsequently initiated. Note that this DEFINE influences processes and is propagated or suppressed according to the rules for all DEFINES.

- `export _ERASE_ON_FREE_={ON|OFF}`

The environment variable \_ERASE\_ON\_FREE\_ can enable or disable the overwrite released-space feature, and it takes precedence over the DEFINE \_ERASE\_ON\_FREE\_ when both are active.

## Using the Programmatic Heap-Management Attributes

Both the original and the high-performance heap managers enable you to set and query certain attributes. [Table 2-5, Heap-Management Attributes for the High-Performance Heap Manager](#), on page 2-46, lists the attributes, their default values, and their limits. For example, you can monitor and, to some extent, control the allocation of flat extended segments performed by the heap manager. [Table 2-5](#) lists the attributes in the format used for pTAL and declared in the RTLDECS file. The format used for C and C++, which is declared in the RTLDECH file, uses underscores (\_) instead of circumflexes (^) in the attribute names.

To query and set heap-management attributes, you use the `RTL_heap_getattribute_` and the `RTL_heap_setattribute_` procedures, which are defined in the RTLDECS file for pTAL, and in the RTLDECH file for C and C++.

# Setting Heap-Management Attributes

You can set five heap attributes: minimum block, erase on free, erase on get, segment threshold, and segments max.

The C syntax of the routine for setting a heap-management attribute is:

```
int RTL_heap_setattribute_ ( int Attribute, size_t Value ) ;
```

*Attribute* is an ordinal that specifies the heap attribute. [Table 2-5](#) on page 2-46 defines the ordinals for *Attribute*.

*Value* supplies the value to be assigned to the specified attribute.

The routine returns 0 if the call succeeds, or 1 if *Attribute* is not recognized, or 2 if *Value* (interpreted as a signed integer) is negative.

**Table 2-5. Heap-Management Attributes for the High-Performance Heap Manager**  
(page 1 of 2)

Attribute Name	Definition and Default Value	Attribute Value
RTL^Heap^min^block	Minimum size in bytes of a block allocated from the heap space. Can be used to prevent accumulation of numerous unusable chunks of heap space.	0D
RTL^Heap^erase^on^free	Zero = Off Nonzero = On (The heap manager overwrites the contents of released space with repetitions of the hexadecimal value FFFC3C3C.) Can be used to identify programs that erroneously use data from freed blocks.	1D
RTL^Heap^erase^on^get	Zero = Off Nonzero = On (The heap manager initializes the contents of allocated space with repetitions of the hexadecimal value FFFC2B2B.) Can be used to identify where a program fails to initialize dynamically allocated space.	2D
RTL^Heap^segment^threshold	The size in bytes beyond which the heap manager can optionally allocate flat extended segments. Below this size, the heap manager will not allocate flat or external segments.  Default value: 16384 * 2048 (33,554,432)	3D



**Table 2-5. Heap-Management Attributes for the High-Performance Heap Manager**  
(page 2 of 2)

Attribute Name	Definition and Default Value	Attribute Value
RTL^Heap^segments^max	The number of flat extended segments that can be allocated subsequently. Default value: 0 Range of values: 0 to 32.	4D
RTL^Heap^space^size	The heap space area size in bytes.	10D
RTL^Heap^space^active	The number of bytes in the heap space area that are currently active, that is, the number of bytes that belong to allocated blocks or to free blocks that reside between allocated blocks.	11D
RTL^Heap^free^blocks	The count of the free space blocks currently present in the heap space area.	12D
RTL^Heap^free^space	The sum of the sizes in bytes of all free space blocks currently present in the heap space area.	13D
RTL^Heap^segments	The number of flat extended segments already allocated.	14D
RTL^Heap^segment^space	The number of bytes in flat extended segments.	15D

## Querying Heap-Management Attributes

You can query the value of all the heap attributes listed in [Table 2-5](#). However, the following six heap attributes can only be queried and not set: space size, space active, free blocks, free space, heap segments, and heap segment space.

The C syntax for the routine that queries heap-management attributes is:

```
int RTL_heap_getattribute_ ( int Attribute, size_t *Value ) ;
```

*Attribute* specifies the heap attribute.

*Value* references the data object to be assigned the attribute value.

## TNS CRE Traps and Exceptions

This subsection describes how the TNS CRE handles run-time errors. It covers:

- Errors in program logic (including arithmetic overflow)
- Hardware traps (except arithmetic overflow)
- Catastrophic errors
- TNS CRE trap handler

- Using ARMTRAP
- Writing messages to standard log
- Handling errors in TNS CRE-supported languages

In the Guardian environment, the TNS CRE performs the tasks described in this subsection. In the OSS environment on G-series systems, the TNS CRE does not process traps or exceptions. Traps are a feature of the Guardian environment. Signals provide an analogous capability in the OSS environment. See the *Open System Services Programmer's Guide* for more information about signals.

In the OSS environment on G-series systems, the TNS CRE reports only non-recoverable errors (for example, a corrupted run-time environment). See [Reporting CRE Errors in the OSS Environment](#) on page 2-54 for more details.

## Errors in Program Logic

An error in your program that is not caused by a trap (except for an arithmetic overflow trap) and does not corrupt the CRE's data or a run-time library's data is considered a program logic error. The following are examples of program logic errors:

- Arithmetic overflow
- Insufficient resources; for example, out of heap space
- Invalid actual parameter passed to a standard function
- Invalid result from a standard function
- I/O error when accessing a file
- Case statement in which the case selector does not match any case alternative, including an "otherwise" alternative

Program logic errors are processed by the run-time library for the routine in which the error occurred.

## Error Handling and Math Standard Functions

When a CRE (or RTL) math function detects an invalid parameter or an error in computing the function's value, the function causes an arithmetic fault to occur when the math function returns to its caller.

In the TNS environment, if your program invoked the math function using a standard C, COBOL, or FORTRAN construct, the run-time library for the routine that invoked the math function determines your program's behavior.

If you invoke a standard math function from a TAL routine in the TNS environment, your TAL routine can determine the effect of the error by setting or resetting the trap-enable bit of the TNS environment register. Ensure that the trap-enable bit of the TNS environment register is set according to the needs of your program before the program calls a math function.

If an error occurs in a math function and traps are disabled in the routine that called the math function, control is returned to the caller:

```
REAL r, s;
r := -1.0E0;
CALL disable_overflow_traps;    ! A user-written routine that
s := RTL_Sqrt_Real32_( r );    ! disables overflow traps
IF $OVERFLOW THEN              ! Control returns here: test
BEGIN                          ! error in Sqrt routine
    ...
END;
```

If traps are enabled when a math function detects an error, the system transfers control to the current trap handler upon return from the function:

```
REAL r, s;
r := -1.0E0;
CALL enable_overflow_traps;    ! A user-written routine that
s := RTL_Sqrt_Real32_( r );    ! enables overflow traps
IF $OVERFLOW THEN              ! Program does not reach this
BEGIN                          ! statement because the
    ...                        ! system transfers control to
END;                           ! the current trap handler
```

## Hardware Traps

Hardware traps are exception conditions detected by the hardware of the processor on which your program is running. The following traps can be reported:

CRE Message Number	Hardware Trap
2	Illegal address reference
3	Instruction failure
4	Arithmetic fault
5	Stack overflow
6	Process loop-timer timeout
7	Memory manager read error
8	Not enough physical memory
9	Uncorrectable memory error
10	Interface limit exceeded

Except for trap 4, arithmetic fault, the CRE treats all of the preceding traps as fatal and terminates your program.

## Catastrophic Errors

The CRE and the run-time libraries maintain data in the upper 32K words of the user data segment and in extended memory. If your program uses a small-memory model, the CRE maintains the user heap in a portion of the lower 32K words of the user data segment.

The CRE and the run-time libraries validate portions of their data at strategic points of execution to ensure that the data values are not only valid, but that they are meaningful in the context in which they are used.

The CRE defines an error in which its data has been corrupted or its logic state has been compromised to be a catastrophic error. If your program has a logic error and writes over CRE data or run-time library data, the CRE or run-time library reports a “Corrupted data” or “Logic error” message. Your program might be using an unchecked array index that exceeds the bounds of its array or your program might be using a pointer that holds an incorrect address.

In addition to possibly corrupting CRE data or run-time library data, your program might be overwriting the pointers to the CRE’s data and to run-time library data. At G[0] and G[1], the CRE maintains pointers to its primary data structure, the Master Control Block (MCB), and to its Heap Control Block (HCB). If your program overwrites G[0]—for example, by using a nil-valued pointer to store data—the CRE reports a “Corrupt MCB” message when the CRE or a run-time library function validates the pointer. Note that run-time library functions might not validate the MCB pointer each time they use it. The result of an operation that uses data referenced by an invalid pointer is undefined.

## TNS CRE Trap Handler

During program initialization, the TNS CRE enables a trap handler in its own domain.

If your program enables its own trap handler and a trap occurs, your trap handler can:

- Attempt to write an error message to standard log.
- Attempt to continue processing.
- Terminate your program.

Either of the first two actions could fail, depending on the nature of the trap.

If the TNS CRE trap handler is in effect when a trap occurs, the operating system transfers control to the TNS CRE trap handler. The TNS CRE trap handler takes the following actions:

- It validates its data. If it finds that its data has been corrupted, the TNS CRE invokes `PROCESS_STOP_`, specifying the `ABEND` option and the message text “Corrupted environment.”
- If the TNS CRE finds that its data is intact, it attempts to write a message to standard log that identifies the trap that occurred. If the TNS CRE cannot write to standard log, it writes the message to the process’s home terminal—unless you specified the `EXECUTION-LOG PARAM`, in which case the TNS CRE does not write an error message.
- If `PARAM INSPECT ON` was set when your program started execution, the TNS CRE calls the `PROCESS_DEBUG_` system procedure.

Although the TNS CRE calls `PROCESS_DEBUG_`, you control whether your program uses the default debugging utility or the symbolic debugging utility by the values you specify for compiler directives, Binder settings, and TACL commands.

See the *Guardian Programmer's Guide* for more information on establishing the debugger program for your process.

- If the TNS CRE cannot write a diagnostic message to standard log, but it can write to your home terminal, it writes message 61, "Standard log file error (*error*)," to your home terminal. *error* appears only if the TNS CRE has an error number it can report. See message 61 in [Section 10, Run-Time Diagnostic Messages](#), for more details.
- If the standard log is available, the TNS CRE writes a stack trace that shows the user routine in which the error occurred as the top element on the stack. If the error occurred in a run-time library, in the TNS CRE, or in a system procedure, the top element of the stack trace is the user routine that invoked the routine in its run-time library, in the TNS CRE, or in the operating system.

---

**Note.** When the CRE prints a stack trace, it scans from the top of the stack toward the bottom of the stack, skipping entries on the top of the stack until it encounters a routine name that does not end in an underscore. The top of the stack trace begins with the first procedure whose name does not end in an underscore. Therefore, if you create a routine whose name ends in an underscore and you call a system procedure from your routine, a stack trace will not include your routine. Therefore, do not create routines whose names end in an underscore.

---

- The TNS CRE calls `CRE_Terminator_` to bring your program to a smooth stop.

## Using ARMTRAP

If your program invokes the ARMTRAP system procedure, it will receive notification of all traps, regardless of the language of the routine that was executing when the trap occurred. After your program has called ARMTRAP, ARMTRAP must handle all traps. It cannot handle traps for a portion of your program and then return trap handling to the TNS CRE. For this reason, do not invoke the ARMTRAP system procedure.

## Writing Messages to Standard Log

The CRE writes all messages to standard log, regardless of the language of the routine that caused the message to be written or how many languages are represented in your program.

The CRE writes most of the messages that appear in standard log on behalf of run-time libraries that support the routines in your process. For example, the TNS CRE writes COBOL messages only as a result of requests it receives from run-time libraries—typically the COBOL run-time library—to do so. The TNS CRE does not write COBOL messages because of errors it detects during its own processing.

Any routine in your process has the ability to send a request to the CRE to write a message to the log file. In general, however, messages identified as COBOL messages are written in response to requests from the COBOL run-time library, and so forth.

Although the TNS CRE does not determine the effects of, or suggest how to recover from, errors that occur in routines that your program invokes—for example, in string functions, math functions, and so forth—the messages in [Section 10, Run-Time Diagnostic Messages](#), give you some guidance. Consult the language manual for the function that caused the error to determine the error's effect and how to recover from the error.

## Language-Specific Error Handling

When an error occurs in your program, its subsequent action depends on the language of the routine that caused the error. This subsection provides a brief overview of error handling for each of the languages supported by the CRE.

### C Routines

If an error occurs in a CRE or run-time library function called from a C environment, the CRE returns control to the C run-time library. The C run-time library either:

- Calls the CRE to write a message to the standard log and then terminates the program.
- Stores a predefined value in the special global variable `errno`, and returns control to your C function.

The CRE or the C run-time library terminates your program unconditionally if:

- The C run-time library finds that its data is invalid or a logic error occurs.
- If you specify the CHECK compiler pragma and a run-time check fails.

- A trap occurs. However, your program can retain control if an arithmetic trap (trap 4) occurs by using the C `trap_overflows()` library function:

```
trap_overflows( enable_flag )
```

---

**Caution.** The C `trap_overflows()` library function takes the following actions:

- It sets or resets the trap enable bit (bit 8) in the TNS environment register.
- It traverses the stack markers in the run-time stack and enables or disables the trap enable bit in each stack marker, according to whether your program calls `trap_overflows` to set or to reset trapping. These actions can have two major effects on your program:

Before a call to `trap_overflows`, some stack markers might have the trap enable bit set, and other stack markers might have the trap enable bit reset. After a call to `trap_overflows`, all stack markers are the same—either set or reset.

If the stack markers of routines that called your C functions include routines written in languages other than C, your program might not behave as you expect. The code emitted by compilers for languages other than C might be based on whether or not traps are enabled. If your C function returns to a routine written in a language other than C, for example a COBOL routine, and the trap enable bit in the stack marker for the COBOL routine has changed, your program might not compute the correct results.

---

If your C function disables traps, enable traps when the function reaches the end of the code that requires that traps be disabled.

The *Guardian TNS C Library Calls Reference Manual* describes the `trap_overflows()` library function.

See the *C/C++ Programmer's Guide* for further details on C error processing.

## COBOL Routines

If an error occurs in a CRE or run-time library function called by the COBOL run-time library, the CRE returns control to the COBOL run-time library, which calls the CRE to write a message to standard log. If your program specifies a declarative for the statement that failed, the COBOL run-time library sets the program's status code and the GUARDIAN-ERR special register, and performs your program's declarative. Your program's behavior following execution of the declarative depends on the error that occurred and the code in the declarative.

If you have not specified a declarative, the COBOL run-time library immediately calls the CRE to terminate your program.

See the *COBOL Manual for TNS and TNS/R Programs* for further details on TNS and TNS/R COBOL error processing. See the *COBOL Manual for TNS/E Programs* for further details on TNS/E COBOL error processing.

## FORTRAN Routines

If an error occurs in a TNS CRE or run-time library function called from a FORTRAN routine, the TNS CRE returns control to the FORTRAN run-time library. If the error occurred when your program executed an I/O statement and you specified either the IOSTAT or ERR parameters on the I/O statement, the FORTRAN run-time library returns control to your FORTRAN program; otherwise, the run-time library terminates the program.

See the *FORTRAN Reference Manual* for further details on FORTRAN error processing.

## TAL Routines

If an error occurs in a CRE function called from a TAL routine, the CRE returns control to the TAL routine without taking any specific action. Because TAL does not have a run-time library, your TAL routine must handle all errors explicitly.

## pTAL Routines

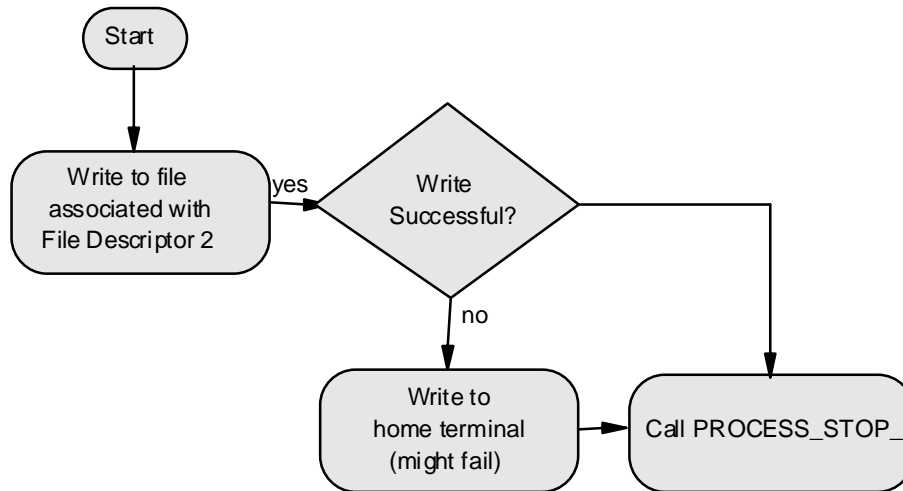
If an error occurs in a CRE function called from a pTAL routine, the CRE returns control to the pTAL routine without taking any specific action. Because pTAL does not have a run-time library, your pTAL routine must handle all errors explicitly.

## Reporting CRE Errors in the OSS Environment

If the CRE detects that the run-time environment is corrupted, it reports an error and terminates the process. The flow chart in [Figure 2-16](#) shows how the CRE determines the file to which it writes error messages. Note that the write to the home terminal might fail. Whether the write to the home terminal succeeds or fails, however, the CRE calls `PROCESS_STOP_`.



---

**Figure 2-16. Writing a CRE Error Message in the OSS Environment**


VST 217VSD

---

## Native CRE Signals and Exceptions

For Guardian processes, the native CRE library provides a default signal handler for all signals whose default action is not signal ignore. The signal handler takes all appropriate actions, then terminates the process; it does not return control to its caller.

The default signal handler is set once at process initialization time. Any subsequent calls to signal manipulation functions by the user can associate a different signal handler with a given signal.

The native CRE library does not provide a default signal handler for OSS processes.

In standard math functions, the native CRE library provides only non-trapping variants of those functions that can cause arithmetic faults. Arithmetic faults are dynamically detected, and the native CRE takes all other appropriate actions, such as setting `errno` and returning an appropriate value.

For details on signals and signal handlers, refer to the *Guardian Programmer's Guide* or *OSS Programmer's Guide*.

# Using CRE Services

**Table 2-6. TNS CRE Services Available in the OSS and Guardian Environments**

Function or Function Group	OSS Environment (G-Series Systems)	Guardian Environment
CRE Standard Files	No	Yes
\$RECEIVE	Limited	Yes
CRE_Terminator_	Yes	Yes
Exception (Trap) Handling	No	Yes
Process Pairs	No	Yes
Signal Handling	Yes	No

**Table 2-7. Native CRE Services Available in the OSS and Guardian Environments**

Function or Function Group	OSS Environment	Guardian Environment
CRE Standard Files	No	Yes
\$RECEIVE	Limited	Yes
CRE_Terminator_	Yes	Yes
Exception (Trap) Handling	No	No
Process Pairs	No	Yes
Signal Handling	Yes	Yes

## Using Standard Functions

The CRE provides a library of standard functions that are required by the language-specific run-time libraries to support your program. The CRE library functions define the same semantics as those used in C-series run-time libraries, including the slight variations that are defined by some of the languages.

The CRE ensures that if an error occurs, function behavior is consistent within a language and, as much as possible, across languages.

For example, the SQRT function always uses the same rules to validate its parameter's domain, regardless of the language from which the SQRT function is called.

The format of the error message that the CRE writes to standard log if the SQRT parameter is not within the proper domain (that is, if it is less than zero) has the same format as all other messages written to standard log.

TNS CRE library functions are intended for the use of the C, COBOL, and FORTRAN run-time libraries, not user-written C, COBOL, or FORTRAN routines. Likewise, the

native CRE library functions are intended for the use of the C and C++ run-time libraries, not for use by user-written C, C++, and pTAL routines.

Only make explicit calls to the CRE library when the language's semantics do not provide the desired CRE service. The ability to make explicit calls to CRE library functions is intended primarily for TAL and pTAL routines in programs written in more than one programming language—mixed-language programs—that must interact with resources managed by the CRE.

**Table 2-8. TNS CRE Standard Functions Available in the OSS and Guardian Environments**

Function or Function Group	OSS Environment (G-Series Systems)	Guardian Environment
Standard Math	Yes	Yes
Sixty-Four Bit Logical Operators	Yes	Yes
Decimal Conversion	Yes	Yes
String	Yes	Yes
Memory Block	Yes	Yes
SMU (CLU Library)	No	Yes
Environment Information	Yes	Yes

**Table 2-9. Native CRE Standard Functions Available in the OSS and Guardian Environments**

Function or Function Group	OSS Environment	Guardian Environment
Standard Math	Yes	Yes
Sixty-Four Bit Logical Operators	Yes	Yes
Decimal Conversion	No	No
String	Yes	Yes
Memory Block	Yes	Yes
SMU (CLU Library)	No	Yes
Environment Information	Yes	Yes

## CRE and RTL Prefixes

Each TNS CRE standard function begins with a four-character prefix: either CRE\_ or RTL\_. Some functions have only a CRE\_ version, others have only an RTL\_ version, and still others have both a CRE\_ version and an RTL\_ version.

For some standard functions, the native CRE provides only the function names, which are case-sensitive. Tables are provided at the beginning of each function or function group section, showing which functions the native CRE supports and the

corresponding function name. See the *Guardian Native C Library Calls Reference Manual* for descriptions of these functions.

## CRE\_ Functions

A function whose name begins with `CRE_` is integral to the CRE. For example, the CRE provides services that manage I/O, `$RECEIVE`, and traps. These are the services that the CRE provides. `CRE_` functions often require access to information from previous CRE operations and save information that is used during subsequent calls to the CRE.

## RTL\_ Functions

Functions whose names begin with the `RTL_` prefix are included in the CRE for one of the following reasons:

- They support functionality that is required by one or more language-specific run-time libraries.
- They are context-free: their parameters specify all of their input values and they return their results in their return values and in reference parameters. They do not depend on calls to the operating system, to the CRE, or other environments.
- They guarantee that they will return a result without causing an exception or an error.

For example, the sine function is guaranteed to return a result regardless of the parameter that you pass to it and does not depend on previous calls made to it.

- A function needs to be available but there are no other libraries in which to store it for general access from any HP language.

## Functions With CRE\_ and RTL\_ Names

Some functions have two definitions that are identical except that one has a `CRE_` prefix and the other an `RTL_` prefix. The CRE supports the `RTL_` versions for compatibility with C-series object files and for D-series object files that do not use the CRE. (See [Section 3, Compiling and Binding Programs for the TNS CRE](#), for more details on support for C-series programs.) The `CRE_` version provides optimal support for those languages and libraries that need to take full advantage of the services of the CRE.

For example, your program can call either of the following functions:

```
CRE_Arccos_Real32_  
RTL_Arccos_Real32_
```

If the parameters you pass do not cause an error, the effect of calling either of these functions is identical. If an error occurs while the function is computing the return value, however—for example, an arithmetic overflow or a domain error—the `CRE_` version sets up the `errno` field required for C functions, and saves the number of the error that

occurred. The RTL\_ version does not save the *errno* field nor does it save the number of the error that occurred.

## Type Suffixes

Many standard functions are represented by multiple versions, depending on the types of the function's parameters and the result returned by the function. For example, you can invoke either `RTL_Sin_Real32_` (which accepts a 32-bit REAL value and returns a 32-bit REAL result) or `RTL_Sin_Real64_` (which accepts a 64-bit REAL value and returns a 64-bit REAL result). The suffix corresponds to the types of the operands and the results returned by the function. Use the function that corresponds to the data type of your data.

## Using Process Pairs

Both the TNS CRE and the native CRE support process pairs for fault-tolerant programming. A process pair is a logical process that runs as two physical processes, each in a separate processor. One of these processes is the primary process and the other is the backup process. If the primary process becomes unavailable, the backup process takes over and continues running.

In the Guardian environment, the CRE performs the tasks described in this subsection. The OSS environment, however, does not support process pairs.

The primary process performs all of the computations specified by your program and the run-time functions that your program invokes. While the primary process successfully runs, the backup process does not execute any of your program's instructions. It merely stores state information passed to it by the primary process so that if the primary process fails, the backup process has the information it needs to become the primary process.

In addition to executing your program's instructions, the primary process also:

- Creates the backup process.
- Sends information at strategic points of its execution—called checkpoints—to the backup process so that if the primary process becomes unavailable (for example, its processor fails, or a logic error in your program causes the primary process to fail), the backup process can take over processing and assume the role of the primary process.
- Monitors `$RECEIVE` for messages from the backup process that report the status of previously sent checkpoint messages.
- Starts a new backup process if the backup process becomes unavailable.

## Requirements for Using Process Pairs

The CRE runs your program as a process pair only if your program meets the following requirements:

- The program's main routine must specify that it can run as a process pair. The routine's compiler sets this attribute in your program's object file based on how you set the NONSTOP compiler directive.
- Your TACL environment must not specify PARAM NONSTOP OFF. You do not need to specify PARAM NONSTOP ON because ON is the default value for the NONSTOP parameter.
- The program must invoke a statement in the language of your main routine that initiates creation of a backup process.
- If the program opens \$RECEIVE, it must open it in TNS COBOL or FORTRAN, or use routines written in TAL to call TNS CRE functions that manage \$RECEIVE. The program cannot open \$RECEIVE from a C routine nor can it open \$RECEIVE by calling the FILE\_OPEN\_ system procedure if you want the TNS CRE to manage backup processing for your program.

The CRE and the TNS COBOL and FORTRAN run-time libraries ensure that the backup process receives not only the data that you checkpoint, but all of the CRE and run-time library data that the backup process requires to take over as the primary process if the current primary process cannot continue running.

## Language Support for Process Pairs

To program using process pairs, you must use either the FORTRAN or the COBOL language. To run as a process pair, your program must use COBOL or FORTRAN constructs to start a backup process and ensure that the backup process has the information required to become the primary process if the current primary process is unable to continue executing.

For example, a routine written in COBOL starts a backup process by executing a STARTBACKUP statement. The COBOL run-time library, with support from the CRE, creates a backup process. The COBOL program itself does not participate further in starting the backup process until the STARTBACKUP statement completes. Upon completion, the COBOL run-time library sets a status code. Test the status code in your COBOL routine to determine the result of the STARTBACKUP operation. Your program uses checkpoint statements in the appropriate language (COBOL or FORTRAN) at strategic points of its execution to send checkpoint information to its backup process.

In mixed-language situations, write all the routines that control process pairs, such as STARTBACKUP and CHECKPOINT, either in COBOL or FORTRAN. Do not control process pairs from both COBOL and FORTRAN routines; however, a program that runs with a backup process can contain both COBOL and FORTRAN routines.

## Using C Routines in Process Pairs

Although the COBOL and FORTRAN run-time libraries support process pairs, the C run-time library supports only active backup processes (and thus is incompatible

with CRE support for process pairs). Therefore, a process pair can include C routines only if those routines do not depend on run-time library resources. That is, C routines:

- Cannot request or return heap space.

---

**Note.** A program that runs as a process pair cannot use the heap because only C routines can allocate space from the heap but C is prohibited from allocating heap space in a process pair.

---

- Cannot call system procedures that depend on the operating system environment, such as FILE\_OPEN\_, FILE\_CLOSE\_, READ, or WRITEREAD. You can call system procedures only if the procedures do not depend on prior context and do not write data that is accessed during subsequent calls to system procedures. For example, your program might call system procedures such as NUMIN and NUMOUT.
- Cannot initiate checkpoint operations.

## C Data in a Process Pair

In a process pair, C routines can use data that is:

- Passed as parameters
- Declared locally within the routine
- Declared globally by COBOL or FORTRAN routines

For more details on using process pairs with the TNS CRE, see the *COBOL Manual for TNS and TNS/R Programs* and the *FORTRAN Reference Manual*.

## Results of Operations That Support Process Pairs

Each operation that supports process pairs generates a status code that is made available to your program in a language-specific fashion. [Table 2-10](#) on page 2-61 shows the status codes that can occur.

---

**Table 2-10. Status Codes Returned by CRE Functions That Support Process Pairs** (page 1 of 2)

Status Code	Meaning
0000	The requested operation completed successfully.
0100	A takeover occurred because the primary process stopped.
0101	A takeover occurred because the primary process aborted.
0102	A takeover occurred because the primary process's CPU failed.
0103	A takeover occurred because the primary process called CHECKSWITCH.
1000	The backup process's CPU is down.
2nnn	The CRE is unable to communicate with the backup process. <i>nnn</i> is a file system error code.*

---

\*The CRE converts file system error codes that are greater than 900 to 000.

---

**Table 2-10. Status Codes Returned by CRE Functions That Support Process Pairs** (page 2 of 2)

Status Code	Meaning
3nnn	A call to FILE_OPEN_CHKPT_ failed. nnn is a file system error code.*
4nnn	PROCESS_CREATE_ failed. nnn is a file system error code.*
	Status codes between 4900 and 4999 represent the value returned by the PROCESS_CREATE_ system procedure. The status code is established by adding the number 4900 to the value returned by PROCESS_CREATE_. See PROCESS_CREATE_ in the <i>Guardian Procedure Calls Reference Manual</i> for more information on error codes returned by PROCESS_CREATE_.
5000	There have been more than 10 failures by your program's backup process. The CRE does not attempt to start another backup process.
6000	An invalid parameter or other logic error was detected.
*The CRE converts file system error codes that are greater than 900 to 000.	

## Using the Inspect, Native Inspect, and Visual Inspect Symbolic Debuggers With CRE Programs

This subsection describes how to use the Inspect, Native Inspect, and Visual Inspect symbolic debuggers with programs that use the CRE, especially to locate where a program is overwriting TNS CRE data.

The CRE and run-time libraries report several of the diagnostic messages shown in [Section 10, Run-Time Diagnostic Messages](#) if they find that their data has been corrupted. Some of the messages you might see as a result of run-time library or CRE data corruption are:

Message Number	Message Text
11	Corrupted environment
12	Logic error
13	MCB pointer corrupt
15	Checkpoint list inconsistent
32	Invalid heap or heap control block
35	Heap corrupted

If the CRE or a run-time library reports a “Corrupted data” message, you can use any symbolic debugger appropriate for the type of code to help isolate where the data is being corrupted.



## Selecting a Debugger

You can use the Inspect, Native Inspect, or Visual Inspect symbolic debugger to debug a program that uses the CRE.

- To specify Inspect or Native Inspect as the debugger, run your program using the RUND TACL command; for example:

```
RUND mtro ...
```

- To specify Visual Inspect as the debugger, use the RUNV TACL command, as follows:

```
RUNV mtro ...
```

When you run your program using the RUND or RUNV TACL command, your program enters the appropriate debugger before it executes any of your program's instructions.

## Considerations for Using Inspect or Native Inspect

If you are using Inspect with the CRE, note the following considerations:

- Because the CRE provides multiple connections to a single open of an operating system file, an Inspect INFO OPENS command shows only one open for each standard file, regardless of how many connections the CRE has granted for the file. If none of your routines have the standard file open, the file does not appear in the file list displayed by Inspect.
- When you use the Inspect debugger to display your program's identifiers, it attempts to locate the identifier you specify, regardless of the language of the routine you are currently inspecting. For example, if you are inspecting a C routine, you can display the TAL variables VAR\_A and VAR^B by entering:

```
display var_a, var^b
```

The Inspect debugger locates identifiers whose names you specify in a display command unless it cannot resolve an ambiguity. For example, if your program contains a C and a TAL variable named VAR\_A and the language of the current scope is C, Inspect finds the C variable when you enter:

```
display var_a
```

If you want to display the TAL identifier, you must use the SET LANGUAGE command as follows:

```
SET LANGUAGE TAL
display var_a
```

- You cannot mix identifiers from different languages within an expression in an Inspect display command.

## Considerations for Using Visual Inspect

If you are using Visual Inspect with the CRE, note the following considerations:

- Visual Inspect can find a global variable of any language independent of the language of the scope in the Program Control View. To find a global identifier, open a Display dialog and then select the Global radio button. Type your variable or expression in the Expression box and click OK.

First Visual Inspect looks in the globals associated with the language of the current scope. Next, Visual Inspect looks in the other globals space.

- If you are debugging a mixed C and TAL program that has the same named global defined in C and TAL, and if the current execution point is in C, Visual Inspect finds the C global.
- If the current execution point is in TAL, Visual Inspect finds the TAL global.
- If the global exists in only one language, and other language is the current scope, Visual Inspect finds the one global that exists.
- Visual Inspect can find the correct identifier in the case of an ambiguity.
  - If your program contains both a C and a TAL variable named VAR\_A, Visual Inspect finds the TAL identifier if the language of the source in the current window is TAL.
  - If the source of the current window is C, and you want to find the TAL identifier, you should either use the Display dialog and specify the scope you want, or open a scope view of any TAL routine and create a watch item from that window.
- Because the CRE provides multiple connections to a single open of an operating system file, the Visual Inspect Opens Manager shows only one open for each standard file, regardless of how many connections the CRE has granted for the file. If none of your routines have the standard file open, the file does not appear in the file list displayed by Visual Inspect.
- You cannot mix identifiers from different languages within an expression in the Visual Inspect Expression box.

## Locating the Corrupter of TNS CRE Pointers

If the TNS CRE reports error 13, “MCB pointer corrupt,” you can use the Inspect or Visual Inspect debugger to help locate the statement that is overwriting the MCB pointer.

### Using Inspect

By using low-level Inspect commands, you can set a memory breakpoint at location G[0]. If your program writes data at the memory location at which you have set a

memory access breakpoint, the debugger displays a message that shows which statement overwrote the data.

[Example 2-1](#) on page 2-65 shows the source code of a C program. The statement at line 7 of the program overwrites the TNS CRE's pointer at G[0] to the MCB.

[Example 2-2](#) on page 2-65 shows the information written to standard log when the program is run.

[Example 2-3](#) on page 2-66 is an Inspect session that shows the statement that overwrites the MCB pointer by setting a memory-access breakpoint at location zero. The program breaks after it executes the statement at line 7.

---

**Example 2-1. C Program That Overwrites the MCB Pointer**

```
1. #pragma runnable
2. #pragma symbols
3. main ()
4. {
5.     int *i;
6.     i = 0;
7.     *i = 0100;
8.     *i = 0101;
9.     *i = 0102;
10. }
```

---

---

**Example 2-2. Run of C Program That Overwrites the MCB Pointer**

```
mysubv 45> mtro
PID: 1,148 \ASYS.$FACE.MYSUBV.MTRO
\ASYS.$:1:148:61677064 - *** Run-time Error 013 ***
\ASYS.$:1:148:61677064 - MCB pointer corrupt
\ASYS.$:1:148:61677064 - From _MAIN + %24, UC.00
ABENDED: 1,148
3: Premature process termination with fatal errors or
diagnostics
```

---

---

### Example 2-3. Inspect Session for C Program That Overwrites the MCB Pointer

---

```

mysubv 46> rund mtro                                <-- Run with Inspect.
INSPECT - Symbolic Debugger - T9673D10 - (08JUN92)   System \ASYS
Copyright HP Computers Incorporated 1983, 1985-1992
INSPECT
175,03,00147 MTRO #_MAIN.#10.001(MTR)
-MTRO-b #main                                         <-- Set breakpoint
Breakpoint created: 1 Code #main.#4(MTR) at start user code.
-MTRO-resume                                           <-- Resume execution.
PID: 3,147 \ASYS.$FACE.MYSUBV.MTRO
INSPECT BREAKPOINT 1: #main
175,03,00147 MTRO #main.#4(MTR)
-MTRO-source on                                       <-- Display source code
SOURCE mode has been enabled                         when program reaches
-MTRO-source                                           breakpoint.
    #1          #pragma runnable
    #2          #pragma symbols
    #3          main ()
1 *#4          {                                     <-- Asterisk means
    #5              int *i;                         statement 4 (actually
    #6              i = 0;                          statement 6) is the
    #7              *i = 0100;                      next statement to
    #8              *i = 0101;                      execute.
    #9              *i = 0102;
    #10         }
-MTRO-low                                             <-- Use low-level Inspect.
_MTRO_bm 0,w                                         <-- Set memory breakpoint
_MTRO_high                                           <-- Use high-level Inspect
-MTRO-resume                                         <-- Resume execution
INSPECT MEMORY ACCESS BREAKPOINT
175,03,00147 MTRO #main.#8(MTR)
    *#8          *i = 0101;
-MTRO-source                                         <-- Display full text
1 #4          {                                     showing asterisk on
    #5              int *i;                         statement 8, which
    #6              i = 0;                          shows that the error
    #7              *i = 0100;                      occurred at
    *#8          *i = 0101;                      statement 7.
    #9              *i = 0102;
    #10         }
-MTRO-low                                             <-- Use low-level Inspect
_MTRO_d 0                                           <-- Value at location 0 is
000000: 000100                                     <-- 100 octal, which was
_MTRO_stop                                          <-- stored at statement 7.
mysubv 47>

```

---

## Using Visual Inspect

In Program Control View, choose Breakpoints from the View menu. In the Breakpoints Manager window, choose the Data tab and click the + tool button. From the Add Breakpoint window, select the Address radio button and enter %0 in the Address box. Then click OK.

## Circumventing the CRE

All HP languages support syntax that you can use to circumvent the CRE or to circumvent your languages' run-time libraries by calling system procedures directly. Although some situations might require that you call system procedures directly, do so

with great caution. If you call system procedures to modify a resource that the CRE is managing, for example, a standard file, the CRE cannot ensure that subsequent access to the resource will produce the results you expect.

For standard files, CRE functions provide most of the functionality you need. The CRE includes functions that execute an operating system CONTROL operation (`CRE_File_Control_`), a SETMODE operation (`CRE_File_Setmode_`), and so forth. If you call a system procedure that leaves a file in a state that conflicts with the state known to the CRE, the CRE might or might not detect that its state is inconsistent with that of the operating system for the same resource, or the CRE might process subsequent requests but the results might be incorrect.

Except where absolutely necessary, use the standard constructs of each programming language to access resources that the CRE manages.



# Compiling and Binding Programs for the TNS CRE

Read this section for information about compiling and binding TNS and accelerated programs that run in the Common Run-Time Environment (CRE). This section describes both the Guardian and OSS environments, but focuses on the Guardian environment. Refer to the *C/C++ Programmer's Guide* and *COBOL Manual for TNS and TNS/R Programs* for information about compiling and binding in the OSS environment on G-series systems.

Refer to [Section 4, Compiling and Linking Programs for the Native CRE](#), for details on compiling and linking native programs.

---

**Note.** TNS object code, accelerated object code, and native object code cannot be mixed in one program file. Thus, you cannot mix modules compiled by a TNS compiler or processed by the Accelerator or Object Code Accelerator with modules compiled by a TNS/R or TNS/E native compiler.

---

Before you can use the information in this section, you must convert your existing C-series file system programs to meet the requirements of the D-series file system procedures. Refer to the *Guardian Application Conversion Guide* for detailed conversion information.

## Compiling Programs for the CRE

There are two major changes in compiling programs for the CRE as opposed to compiling programs that do not use the CRE. To compile a program for the CRE, you might need to:

- Specify the desired run-time environment with an ENV compiler directive
- Use a SOURCE directive to specify the external declarations for CRE library functions that are called from TAL routines

---

**Note.** Compiler directives are called “pragmas” in the C language. This manual uses the term “directive” to refer both to directives and to pragmas.

---

## Specifying a Run-Time Environment

You use the ENV directive to specify the run-time environments in which a program can run. For each of the ENV options, the compilers generate different code and impose different restrictions on which run-time libraries and language features routines can use.

[Table 3-1](#) on page 3-2 shows the availability of run-time libraries, depending on the specified ENV option.

**Table 3-1. ENV Options and the Availability of Run-Time Libraries**

ENV Option	Available Run-Time Library
COMMON	CRE library
OLD	A COBOL and FORTRAN run-time library
LIBRARY	CRE library
NEUTRAL	None
EMBEDDED	None
LIBSPACE	None

All language features are available to routines that run in user code space. The ENV directive restricts the availability of certain language features to enable routines to run in user library and system library code spaces. [Table 3-2](#) on page 3-2 shows the availability of language features, depending on the specified ENV option:

**Table 3-2. ENV Options and the Availability of Language Features**

Language Feature	OLD	COMMON	LIBRARY	NEUTRAL	EMBEDDED	LIBSPACE
High-level language I/O operations	Yes	Yes	Yes*	No	No	No
User heap operations	Yes	Yes	Yes	No	No	No
Main routine	Yes	Yes	No	Yes**	No	No
Relocatable data blocks	Yes	Yes	No	Yes	Yes	No

\* LIBRARY permits the use of high-level language I/O facilities if direct access to relocatable data blocks is not needed for the operations.

\*\* NEUTRAL cannot be used for a main routine if the program uses run-time library resources.

Use the ENV directive options as follows:

**Table 3-3. Determining Which ENV Options to Use** (page 1 of 2)

ENV Option	Use for
COMMON	C, COBOL, and FORTRAN user code routines that run in the CRE TAL main routines that run in the CRE
LIBRARY	C or COBOL user library routines that run in the CRE



**Table 3-3. Determining Which ENV Options to Use** (page 2 of 2)

ENV Option	Use for
OLD	COBOL or FORTRAN user code routines that use the TNS COBOL or FORTRAN language-specific run-time libraries only
EMBEDDED	C user code routines that do not rely on run-time libraries; intended for subsystems programming
LIBSPACE	C user library or system library routines that do not rely on run-time libraries; intended for systems programming

All routines in a program must be compiled to run in either a language-specific run-time environment or the CRE. Routines compiled with the ENV NEUTRAL, ENV EMBEDDED, and ENV LIBSPACE directives are exceptions to this rule; these routines do not depend on run-time library resources so they can run in either environment.

All routines that run in the user library space (user library routines) must be compiled for the same run-time environment as their callers in the user code space (user code routines). For example, TAL user library routines compiled with the ENV NEUTRAL directive can be called by TAL routines compiled for either a language-specific run-time environment or the CRE. C and COBOL user library routines can be called only by programs that run in the CRE.

[Table 3-4](#) on page 3-3 lists the recognized and default ENV directive options for each language.

**Table 3-4. Recognized and Default ENV Options**

Language	Recognized ENV Directives	Default ENV Option
C	COMMON, EMBEDDED, LIBRARY, LIBSPACE	COMMON
COBOL	COMMON, LIBRARY, OLD	OLD
FORTRAN	COMMON, OLD	OLD
TAL	COMMON, NEUTRAL, OLD	NEUTRAL

The C compilers generate programs that run in the CRE. The TNS COBOL, FORTRAN, and TAL compilers generate programs that can run in either a language-specific run-time environment or the CRE. To produce TNS COBOL, FORTRAN, or TAL programs that run in the CRE, you must compile with the ENV COMMON directive. To produce TNS COBOL, FORTRAN, or TAL programs that run in language-specific run-time environments, you must compile with the ENV OLD directive or without any ENV directive.

You can specify an ENV directive either in a compilation command or in the program source code before any declarations. For example, the following compilation command produces a TAL object file compiled for the CRE:

```
TAL / IN source, OUT listing / ; ENV COMMON
```

## Compiling TAL Programs for the CRE

TAL routines compiled with the ENV NEUTRAL directive can be bound into a program that runs in either the CRE or a language-specific run-time environment. However, a TAL routine compiled with the ENV NEUTRAL directive cannot be the main procedure in a program that runs in the CRE because the TAL compiler only allocates and initializes the special data blocks required by the CRE when the main routine is compiled with the ENV COMMON directive.

For a program with a TAL main procedure to run in the CRE, the program must make explicit calls to CRE functions for initialization, input/output, and so on. In all the other languages, the CRE is almost invisible to you. Refer to the *TAL Programmer's Guide* for details on writing TAL programs that use the CRE.

## Sourcing-in CRELIB Function Declarations

CRELIB contains the CRE library functions documented in [Section 5, Using the Common Language Utility \(CLU\) Library](#), [Section 6, CRE Service Functions](#), and [Section 7, Math Functions](#). If your TAL routines call CRE library functions whose names begin with CRE\_, you must compile the routines with a SOURCE directive that references CREDECS, the external declarations file for the CRE library functions. Likewise, if your TAL routines call functions whose names begin with RTL\_, you must compile the program with a SOURCE directive that references RTLDECS, the external declarations file for the RTL\_ functions.

CREDECS and RTLDECS are located in \$SYSTEM.SYSTEM by default. Each declaration is placed within a TAL SECTION directive. Some declarations in CREDECS and RTLDECS contain TAL BLOCK statements to declare global data blocks. Your program should read the source code from only those sections of the files that contain declarations you need in your program. Programs that use these declarations must follow the coding guidelines for BLOCK declarations. Refer to the *TAL Programmer's Guide* for details.

## CRE Data Blocks

When a compilation unit contains a main procedure, the C compilers allocate and initialize special data blocks required by the CRE. When a compilation unit contains a main procedure and you have specified an ENV COMMON directive, the TNS COBOL, FORTRAN, and TAL compilers allocate and initialize these special data blocks.

[Table 3-5](#) on page 3-5 lists the special data blocks.

**Table 3-5. Common Run-Time Environment Data Blocks**

Data Block	Name	Description
Basic control block	#CRE_GLOBALS	Located at G[0] and G[1] of the user data segment.
Master control block	#MCB	If a program contains FORTRAN routines, the Binder program allocates the MCB in the upper 32K area of the user data segment, regardless of the value of the HIGHCONTROL FORTRAN directive.
CRE run-time heap	#CRE_HEAP	A run-time heap required by CRE functions. It is the last block in the upper 32K area of the user data segment.

## Binding Programs for the CRE

Binder categorizes the ENV directive parameters into three groups: OLD, NEUTRAL, and COMMON. For the most part, these groups match the various ENV options.

Refer to [Compiling Programs for the CRE](#) on page 3-1 for a complete discussion of the ENV directive. [Table 3-6](#) on page 3-5 shows how Binder classifies object files into one of three groups depending on the compiler version or specified ENV option.

**Table 3-6. Binder Grouping of ENV Directive Parameters** (page 1 of 2)

Binder Group	Language	Generated by
OLD	C	C-series compilers by default
	COBOL	C-series compilers by default D-series compilers by default D-series compilers with ENV OLD specified
	FORTRAN	C-series compilers by default D-series compilers by default D-series compilers with ENV OLD specified
	TAL	D-series compilers with ENV OLD specified
COMMON	C	D-series compilers by default D-series compilers with ENV COMMON specified
	COBOL	D-series compilers with ENV COMMON specified D-series compilers with ENV LIBRARY specified

**Table 3-6. Binder Grouping of ENV Directive Parameters** (page 2 of 2)

Binder Group	Language	Generated by
	FORTTRAN	D-series compilers with ENV COMMON specified
	TAL	D-series compilers with ENV COMMON specified
NEUTRAL	C	D-series compilers with ENV EMBEDDED specified D-series compilers with ENV LIBSPACE specified
	TAL	C-series compilers by default D-series compilers by default D-series compilers with ENV NEUTRAL specified

The rules for binding modules together follow:

- You can bind object files that are in the same Binder group; the resulting object file runs in the same environment as the input object files.
- You can bind object files that include routines from both the OLD and NEUTRAL Binder groups; the resulting object file runs in a language-specific run-time environment.
- You can bind object files that include routines from both the COMMON and NEUTRAL Binder groups; the resulting object file runs in the CRE.
- You cannot bind object files that include routines from both the COMMON and OLD Binder groups.

When you bind object files compiled for different environments, each procedure retains its original ENV attribute.

**Note.** If you specify the ENV NEUTRAL directive in a TAL source file, BINSERV does not allow the resulting object file to be combined with object files compiled for the COMMON or OLD Binder groups. For example, BINSERV will not generate an object file if a file that contains an ENV NEUTRAL directive also specifies a SEARCH directive to a file compiled with ENV COMMON or ENV OLD directives. This rule prevents an object file from gaining the COMMON or OLD attributes.

[Table 3-7](#) shows the run-time environment resulting from binding modules from different Binder groups together.

**Table 3-7. Run-Time Environment Resulting From Binding Modules**

Binder Group	OLD	COMMON	NEUTRAL
OLD	language-specific	Not allowed	language-specific
COMMON	Not allowed	CRE	CRE
NEUTRAL	language-specific	CRE	language-specific or CRE

Use the Binder INFO command with the DETAIL clause to show the ENV attribute of a particular data or code block. For example:

```
@ add * from test
@ info *, detail
```

```
TEST^PROCEDURE                                167
  LANG: TAL      ENV: NEUTRAL    TIME: 1992-06-19  09:48
```

## Run-Time Libraries

The TNS C, COBOL85, FORTRAN, and TAL products each consist of a compiler and a set of separately packaged run-time functions called a run-time library. Compilers generate code that depends on the availability of functions in the run-time libraries. The operating system requires that all references to run-time functions be resolved before a program is run. To resolve these references, you must either use Binder to bind the necessary run-time functions into your program or configure the run-time functions into the system library. The operating system resolves references to run-time functions at process startup for routines in the system library.

[Table 3-8](#) on page 3-7 shows the locations of the run-time libraries.

**Table 3-8. Locations of C-Series and D-Series TNS Run-Time Libraries**

Language	Guardian Run-Time Library	OSS Run-Time Library (G-series)
C	CLIB is in the system library.	
COBOL	C8LIB is in the system library.	
FORTTRAN	FORTLIB is in the subvolume containing the FORTRAN compiler. FORTSYS is in the system library.	N.A.
TAL	TALLIB is in the subvolume containing the TAL compiler.	N.A.

The TNS C, COBOL, and FORTRAN run-time libraries are in the system library. You never have to bind the run-time library into TNS C, COBOL, or FORTRAN programs; however, you must bind the appropriate memory-model file into C programs. Refer to the *C Programmer's Guide* for details on the C memory-model files.

You must bind the TALLIB file into programs with TAL routines that contain embedded SQL statement or calls to the TAL\_CRE\_INITIALIZER\_ procedure. Refer to the *TAL Programmer's Guide* for more information.

The TNS COBOL and FORTRAN run-time libraries include functions that run in the CRE and functions that run in the COBOL and FORTRAN language-specific run-time environments. Thus, TNS COBOL and FORTRAN programs can run in their language-specific run-time environments or the CRE.

In addition, a run-time library called CRELIB contains the CRE library functions documented in [Section 5, Using the Common Language Utility \(CLU\) Library](#),

[Section 6, CRE Service Functions](#), and [Section 7, Math Functions](#). The TNS C, TNS COBOL, and FORTRAN run-time libraries call the functions in CRELIB. If you bind the appropriate languages' run-time libraries into your program, you must also bind in CRELIB.

## Sample Binder Sessions

The following binding example assumes that your program is made up of a TNS C object file (CFILE), a TNS COBOL object file (COBFILE), and a TAL object file (TALFILE). The C object file uses the wide-memory model in C, and the TAL object file contains routines that call CRE library functions.

To create an executable program of minimal size, enter the following Binder commands:

```
ADD * FROM CFILE
ADD * FROM COBFILE
ADD * FROM TALFILE
SELECT SEARCH CWISE -- Add wide memory-model C library functions
BUILD MyProg
```

## Bind-Time Validation for Mixed-Language Programs

To help you bind mixed-language programs, Binder provides parameter, return-value, and language consistency checking.

### Parameter and Return-Value Checking

The Binder SELECT CHECK PARAMETER command specifies the extent to which Binder checks the consistency of parameters and return value types between called and calling routines.

Binder issues a warning message if the called routine's parameter requirements do not match those of the caller. Because each language has its own set of data types, matching parameters and return-value types between languages cannot be exact. To reduce the number of extraneous Binder warnings generated when you bind mixed-language programs, specify the SELECT CHECK PARAMETER STRONG option. (The STRONG option is more lenient than the STRICT option.) Under the STRONG option:

- Formal and actual parameters for intralanguage calls must be the same size, type, and mode (passed by value or by reference).
- Formal and actual parameters for interlanguage calls must belong to the same class. The classes of parameters are:
  - Two-byte scalar, passed by value
  - Four-byte integer scalar, passed by value
  - Four-byte real scalar, passed by value
  - Eight-byte integer scalar, passed by value
  - Eight-byte real scalar, passed by value
  - Byte address

- Word address
- Extended address
- Two-byte procedure parameter
- Four-byte procedure parameter

Only C supports the passing of structured parameters by value. Such parameters cannot match parameters passed from any other language.

- Return types (values returned on the stack) for both interlanguage and intralanguage calls must belong to the same class. The classes of return values are:
  - Two-byte integer scalar
  - Four-byte integer scalar
  - Four-byte real scalar
  - Eight-byte integer scalar
  - Eight-byte real scalar
  - No return type

## Language Consistency Checking

Binder provides language consistency checking by making sure that callers specify the correct language for called routines. If a caller explicitly states that the language of a called routine is unspecified, Binder does not perform this check.

If one caller specifies one language, and a subsequent caller specifies a different language, and neither of them has been resolved, Binder issues the message:

```
**** WARNING 149 **** Referencing procedures do not agree on the
language of procedure procedure-name.
```

After it issues the message, Binder must determine which language to use for subsequent checking. If either caller is written in the same language as the called routine, Binder uses that language. If both callers specify languages other than that of the called routine, Binder selects the language of one of the caller routines. Binder makes this selection in the following order:

TNS COBOL  
 FORTRAN  
 TAL  
 TNS C

For example, if one caller specifies C and another caller specifies COBOL, Binder selects COBOL for the called routine.

Once Binder has determined the language of a called routine, and a caller specifies a different language, Binder issues the message:

```
**** WARNING 148 **** Referencing procedures claim that
procedure procedure-name is written in a different language.
```

If Binder issues these messages during a bind session, examine calls to the named procedure in your source code to make sure they specify the correct language. Note

that two routines in the same object file cannot have the same name, even if they are written in different languages.

Refer to the *Binder Manual* for more information.



# Compiling and Linking Programs for the Native CRE

Read this section for information about compiling and linking native programs that run in the Common Run-Time Environment (CRE). This section describes both the Guardian and OSS environments, but focuses on the Guardian environment. Refer to the *Open System Services Shell and Utilities Reference Manual* and the *C/C++ Programmer's Guide* for information about compiling and linking in the OSS environment.

See [Section 3, Compiling and Binding Programs for the TNS CRE](#), for details on compiling and binding TNS and accelerated programs.

**Note.** TNS object code, accelerated object code, and native object code cannot be mixed in one program file. Thus, you cannot mix modules compiled by a TNS compiler or processed by the Accelerator with modules compiled by a TNS/R or TNS/E native compiler.

Before you can use the information in this section, you must convert your existing C-series programs to meet the requirements of the D-series operating system. Refer to the *Guardian Application Conversion Guide* for detailed conversion information.

Native programs cannot use all of the CRE services available to TNS and accelerated programs. Refer to the *TNS/R Native Application Migration Guide* or the appropriate programming language manual for the changes required to convert TNS programs to native programs.

## Using the Environment Variable for C and C++ Modules

By default, the native C and C++ compilers generate modules that run in the CRE. The `env` pragma (or `-Wenv` flag to the `c89` utility) determines the availability of run-time library and language features in a module. The four `env` pragmas are:

- `env common`
- `env library`
- `env embedded`
- `env libspace`

The default is `env common`. For each `env` pragma, the compilers generate different code and impose different restrictions on which run-time library and language features can be used.

[Table 4-1](#) on page 4-2 shows the features available with each `env` pragma.

**Table 4-1. env Pragma and the Availability of Features**

Feature	env common	env library	env embedded	env libspace
C run-time library and CRE library	Yes	Yes	No	No
User heap operations	Yes	Yes	No	No
Main routine	Yes	No	No	No
Global variable declarations	Yes	No	Yes	No

If an `env library` or `env libspace` pragma is specified, the native C and C++ compilers place literal constants in the read-only data area and generate errors for global variable declarations.

Use the `env` pragmas as follows:

env Pragma	Use for:
<code>env common</code>	User code functions that run in the CRE
<code>env library</code>	User library functions that run in the CRE
<code>env embedded</code>	User code functions that do not rely on run-time libraries; intended for subsystems programming
<code>env libspace</code>	User library functions that do not rely on run-time libraries; intended for systems programming

The `env` pragmas also define four feature-test macros. The feature-test macros are used in HP header files to select the function declarations appropriate for each code space. Feature-test macros are defined as follows:

<code>env common</code>	<code>_COMMON</code>
<code>env library</code>	<code>_LIBRARY</code>
<code>env embedded</code>	<code>_EMBEDDED</code>
<code>env libspace</code>	<code>_LIBSPACE</code>

## Sourcing In CRE External Declarations for pTAL Modules

Unlike TAL routines, a pTAL routine cannot be the main procedure in a program that runs in the CRE. The EpTAL or pTAL compiler does not establish the run-time environment and allocate and initialize the special data blocks required by the CRE. To work around this restriction, write a C main function that simply calls a pTAL routine.

Unlike the TAL compiler, which requires you to specify an `ENV COMMON` or `ENV NEUTRAL` directive for programs that run in the CRE, the EpTAL or pTAL compiler

does not require any ENV directives. The compiler issues a warning for any ENV directives that it finds.

If your pTAL routines call CRE functions, you must compile the routines with a SOURCE directive that references the external declarations file for the CRE function as follows:

If the Function Name Begins With:	Specify This File:
CRE_	CRERDECS
RTL_	RTLREDCS
CLU_	CLURDECS

These external declarations files are located in \$SYSTEM.SYSTEM by default. Each declaration is placed within a pTAL SECTION directive. Some declarations contain pTAL BLOCK statements to declare global data blocks. Your program should read the source code from only those sections of the files that contain declarations you need in your program. Programs that use these declarations must follow the coding guidelines for BLOCK declarations. Refer to the *pTAL Reference Manual* for details.

## Linking Modules

When linking modules to create a program that runs in the CRE, you must specify the correct shared run-time libraries (SRLs) or dynamic-link libraries (DLLs) for the resolution of external references to CRE and native C or COBOL run-time library functions, and in certain cases you must specify an additional library. By default, the OSS and PC workstation `c89`, `nmcobol`, and `ecobol` utilities specify the correct SRLs or DLLs for the CRE.

If you run the `nld` or `ld` utility to perform the linking, you must:

- Always search the CRE SRL file, `ZCRESRL`
- When there are TNS/R native C modules, search the TNS/R native C run-time library SRL file, `ZCRTLSRL`
- When there are TNS/R native COBOL modules, search the TNS/R native COBOL run-time library SRL file, `ZCOBSRL`
- When the main routine is written in TNS/R native C, link in the TNS/R native C run-time library object file, `CRTLMAIN` for a non-PIC file or `CCPPMAIN` for a PIC file

If you run the `eld` utility to perform the linking, you must:

- Always search the CRE DLL file, `ZCREDLL`
- When there are TNS/E native C modules, search the TNS/E native C run-time library DLL file, `ZCRTLDLL`
- When there are TNS/E native COBOL modules, search the TNS/E native COBOL run-time library DLL file, `ZCOBDLL`

- When the main routine is written in TNS/E native C, link in the TNS/E native C run-time library object file, CCPLMAIN

Unlike the Binder product, the native linker utilities do not perform any special checking for mixed-language programs or programs that use the CRE.

## Examples of Compiling and Linking

1. In this example, the pTAL compiler compiles the pTAL source file PTALSRC and generates the object file PTALOBJ. Then the native C compiler compiles the C source file CSRC and generates the object file COBJ. Finally the LD utility links the PTALOBJ and COBJ files with the CRE and C run-time library hybrid SRLs and C run-time library object file and produces the executable file MYEXEC:

```
PTAL / IN PTALSRC, OUT $S.#TEST / PTALOBJ
NMC / IN CSRC, OUT $S.#TEST / COBJ
LD $SYSTEM.SYSTEM.CRTLMAIN PTALOBJ COBJ -O MYEXEC &
  -LIB $SYSTEM.SYS00.ZCRESRL -LIB $SYSTEM.SYS00.ZCRTLSRL
```

Additional DLLs might need to be linked in, depending on the routines your program uses.

The pTAL compiler and the OSS `c89` utility simplify the compiling and linking processes. The same files are used as in the previous example, but the `c89` utility replaces the Guardian environment CCOMP and LD commands, as shown in the following sequence of OSS commands:

```
gtacl -p "ptal / in ptalsrc, out $s.#test / ptalobj"
c89 csrc /G/MYDISK/MYVOL/ptalobj -o myexec
```

2. In this example, the EpTAL compiler compiles the pTAL source file EPTALSRC and generates the object file EPTALOBJ. Then the native C compiler compiles the C source file ECSRC and generates the object file ECOBJ. Finally the ELD utility links the EPTALOBJ and ECOBJ files with the CRE and C run-time library DLLs and C run-time library object file and produces the executable file EMYEXEC:

```
EPTAL / IN EPTALSRC, OUT $S.#TEST / EPTALOBJ
CCOMP / IN ECSRC, OUT $S.#TEST / ECOBJ
ELD $SYSTEM.SYSTEM.CCPLMAIN EPTALOBJ ECOBJ -O EMYEXEC &
  -LIB $SYSTEM.SYS00.ZCREDLL -LIB $SYSTEM.SYS00.ZCRTLDLL
```

Additional DLLs might need to be linked in, depending on the routines your program uses.

The EpTAL compiler and the OSS `c89` utility simplify the compiling and linking processes. The same files are used as in the previous example, but the `c89` utility

replaces the Guardian environment CCOMP and ELD commands, as shown in the following sequence of OSS commands:

```
gtac1 -p "eptal / in eptalsrc, out $s.#test / eptalobj"  
c89 ecsrc /G/MYDISK/MYVOL/eptalobj -o emyexec
```



# Using the Common Language Utility (CLU) Library

This section describes the services provided by the Common Language Utility (CLU) library. It explains how to use the services of the Saved Message Utility (SMU), part of the CLU library, in COBOL, FORTRAN, and TAL routines in the TNS environment, and in C, C++, COBOL, and pTAL routines in a native environment.

CLU library routines can be used only in the Guardian API.

## What Is the CLU Library?

The Common Language Utility (CLU) library is a collection of functions that provides common services to two or more language products. CLU functions provide a separate set of services from the CRE routines.

The CLU provides services that enable:

- COBOL and FORTRAN routines to create processes
- COBOL and FORTRAN routines to locate and identify file connectors (TNS CRE only)
- COBOL, FORTRAN, TAL, and pTAL routines to save and manipulate messages sent to a process by the operating system

CLU functions differ in general from CRE functions as shown in [Table 5-1](#) on page 5-1. Note also that CRE functions work only in the CRE, whereas CLU functions work either in the CRE or in certain language-specific run-time environments.

**Table 5-1. Comparison of CRE and CLU Functions**

	<b>TNS CRE</b>	<b>Native CRE</b>	<b>TNS CLU</b>	<b>Native CLU</b>
<b>Where located</b>	CRELIB file	Shared run-time library, or hybrid dynamic-link library (DLL) for TNS/R DLL for TNS/E	CLULIB file	Shared run-time library, or hybrid dynamic-link library (DLL) for TNS/R DLL for TNS/E
<b>How to use</b>	Can optionally be bound into TNS programs	Binding not required	Must be explicitly bound into TNS programs	Binding not required

# Compiling and Binding or Linking Programs That Use the CLU Library

For the TNS CRE, you must explicitly bind into your program's object file all CLU functions referenced by your program. To bind the CLU functions into your object file, specify the following Binder command:

```
SELECT SEARCH $SYSTEM.SYSTEM.CLULIB
```

To call CLU functions directly, TAL routines must also source-in CLUDECS, the external declarations file for the CLU functions. CLUDECS is located in \$SYSTEM.SYSTEM by default.

For the TNS/R or TNS/E CRE, pTAL routines must source in CLURDECS, a file located by default in \$SYSTEM.SYSTEM.

ECOBOL and NMCOBOL programs can call only the Saved Message Utility functions listed in [Table 5-2](#) on page 5-3. These functions are in the library file ZCRESRL on G-series systems and ZCREDLL on H-series systems.

## Creating Processes

To create a process, routines can use the `CLU_Process_Create_` function. Using this function, a process is created using the conventions of the TACL RUN command. This function is available to any CRE program and to COBOL or FORTRAN programs compiled for execution in a non-CRE environment—that is, with ENV OLD specified or assumed by default. See the definition of this routine in [Section 9, Common Language Utility \(CLU\) Library Functions](#), for more details.

## Locating and Identifying File Connectors

A file connector is an abstract entity through which a program accesses a file. It is physically represented by a run-time data object called a File Control Block (FCB). Each file connector has a logical file name and a physical file name (called the Guardian file name or TANDEMNAME).

To locate and identify file connectors, COBOL and FORTRAN routines can use the `CLU_Process_File_Name_` function. Using this function, your routine can locate a file connector, obtain its identity, and optionally alter the Guardian file name value. See the definition of this routine in [Section 9, Common Language Utility \(CLU\) Library Functions](#), for more details.

The native CRE does not support functions that locate and identify file connectors.

## Using the Saved Message Utility Functions

The Saved Message Utility (SMU) functions listed in [Table 5-2](#) on page 5-3 enable routines to manipulate saved startup, ASSIGN, and PARAM messages that are sent to



your process by the process that starts your process. The SMU functions can be called by COBOL, FORTRAN, TAL, and pTAL-compiled programs.

The TNS CRE SMU functions, data, and data structure declarations in TAL are available in the CLUDECS file. The TNS/R and TNS/E native CRE SMU functions, data, and data structure declarations in pTAL are available in the CLURDECS file.

**Table 5-2. SMU Functions**

<b>Name</b>	<b>Action</b>
<a href="#">SMU_Assign_CheckName</a> on page 9-18	Checks whether an ASSIGN message with a given logical file name exists.
<a href="#">SMU_Assign_Delete</a> on page 9-19	Deletes a portion or all of an ASSIGN message.
<a href="#">SMU_Assign_GetText</a> on page 9-21	Retrieves a portion of an ASSIGN message as text and assigns it to a string variable.
<a href="#">SMU_Assign_GetValue</a> on page 9-22	Retrieves a portion of an ASSIGN message as an integer and assigns it to an integer variable.
<a href="#">SMU_Assign_PutText</a> on page 9-23	Creates or replaces a portion of an ASSIGN message with text from a string variable.
<a href="#">SMU_Assign_PutValue</a> on page 9-25	Creates or replaces a portion of an ASSIGN message with a value from an integer variable.
<a href="#">SMU_Message_CheckNumber</a> on page 9-26	Checks whether a specific message exists.
<a href="#">SMU_Param_Delete</a> on page 9-27	Deletes a portion or all of the PARAM message.
<a href="#">SMU_Param_GetText</a> on page 9-28	Retrieves a portion of the PARAM message as text and assigns it to a string variable.
<a href="#">SMU_Param_PutText</a> on page 9-29	Creates or replaces a portion of a PARAM message with text from a string variable.
<a href="#">SMU_Startup_Delete</a> on page 9-30	Deletes the entire startup message.
<a href="#">SMU_Startup_GetText</a> on page 9-31	Retrieves a portion of the startup message as text and assigns it to a string variable.
<a href="#">SMU_Startup_PutText</a> on page 9-33	Creates or replaces a portion of the startup message with text from a string variable.

TNS COBOL and FORTRAN routines can continue to use pre-D20 SMU functions. [Table 5-3](#) on page 5-4 lists the pre-D20 SMU functions and the corresponding current SMU functions.

**Table 5-3. Pre-D20 and Current SMU Functions**

Pre-D20 SMU Function	Current SMU Function
ALTERPARAMTEXT	None
CHECKLOGICALNAME	<a href="#">SMU_Assign_CheckName</a> on page 9-18
CHECKMESSAGE	<a href="#">SMU_Message_CheckNumber</a> on page 9-26
CREATEPROCESS	None
DELETEASSIGN	<a href="#">SMU_Assign_Delete</a> on page 9-19
DELETEPARAM	<a href="#">SMU_Param_Delete</a> on page 9-27
DELETESTARTUP	<a href="#">SMU_Startup_Delete</a> on page 9-30
GETASSIGNTEXT	<a href="#">SMU_Assign_GetText</a> on page 9-21
GETASSIGNVALUE	<a href="#">SMU_Assign_GetValue</a> on page 9-22
GETBACKUPCPU	None
GETPARAMTEXT	<a href="#">SMU_Param_GetText</a> on page 9-28
GETSTARTUPTEXT	<a href="#">SMU_Startup_GetText</a> on page 9-31
PUTASSIGNTEXT	<a href="#">SMU_Assign_PutText</a> on page 9-23
PUTASSIGNVALUE	<a href="#">SMU_Assign_PutValue</a> on page 9-25
PUTPARAMTEXT	<a href="#">SMU_Param_PutText</a> on page 9-29
PUTSTARTUPTEXT	<a href="#">SMU_Startup_PutText</a> on page 9-33

Pre-D20 and current SMU functions work somewhat differently. For example, the current SMU functions do not support checkpointing. To capture changes in internal run-time data structures for checkpointing purposes, TNS COBOL and FORTRAN routines can use the pre-D20 SMU functions. [Section 9, Common Language Utility \(CLU\) Library Functions](#), describes the current SMU functions.

## Services Provided by the Saved Message Utility

When TACL starts a process, it sends a series of messages to the process that describes the following:

- IN and OUT file names
- Default volume and subvolume
- Current ASSIGN values
- Current PARAM values
- Additional text specified with the RUN command

To save these messages for manipulation with SMU functions:

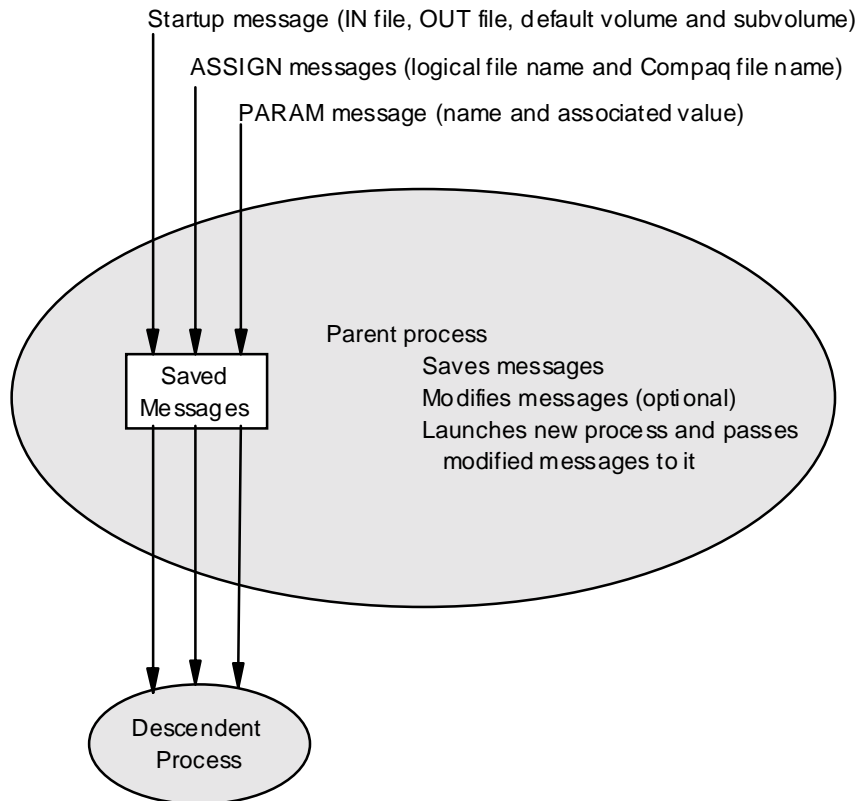
- COBOL and FORTRAN programs use the SAVE compiler directive
- TAL programs call `TAL_CRE_INITIALIZER_` and pass the `OPTIONS` parameter to it, as described in the *TAL Programmer's Guide*.

If a process (a “parent” process) initiates other processes, it can use SMU functions to customize the startup, ASSIGN, and PARAM messages it received when it started, and pass the customized messages to the processes it initiates.

If a process initiates other processes, the parent process can save the messages that describe its startup environment, and then use SMU functions to customize the startup environment for the new processes. [Figure 5-1](#) on page 5-5 illustrates how descendent processes can inherit customized messages from the parent process.

---

**Figure 5-1. Messages Manipulated by the SMU**



VST 501.VSD

SMU functions operate on copies of the process creation messages that establish the execution environment for a program. The functions let you check, retrieve, delete, or replace portions of messages. [Table 5-4](#) on page 5-6 lists the SMU functions by action and by type of message operated on.

**Table 5-4. Using SMU Functions**

Action	Startup Message	ASSIGN Messages	PARAM Message
Check	SMU_Message_CheckNumber_	SMU_Message_CheckNumber_ SMU_Assign_CheckName_	SMU_Message_CheckNumber_
Get	SMU_Startup_GetText_	SMU_Assign_GetText_ SMU_Assign_GetValue_	SMU_Param_GetText_
Put	SMU_Startup_PutText_	SMU_Assign_PutText_ SMU_Assign_PutValue_	SMU_Param_PutText_
Delete	SMU_Startup_Delete_	SMU_Assign_Delete_	SMU_Param_Delete_

## Content of Messages

The content of startup, ASSIGN, and PARAM messages are listed in the following paragraphs.

### Startup Message

The startup message contains information about the startup of process. [Table 5-1](#) on page 5-1 lists the parts of the startup message:

**Table 5-5. Startup Message Parts**

Portion Name	Type	Identifies
VOLUME	Text	Default volume and subvolume names
IN	Text	Input file name
OUT	Text	Output file name
STRING	Text	The startup message's parameter string (the text that follows the RUN option list)

### ASSIGN Messages

ASSIGN messages contain file names and attributes that you specify using a TACL ASSIGN command. [Table 5-6](#) on page 5-6 lists the parts of an ASSIGN message.

**Table 5-6. ASSIGN Message Parts** (page 1 of 2)

Portion Name	Type	Identifies
LOGICALNAME	Text	Logical file name (DEFINE or ASSIGN file name)
TANDEMNAME	Text	Physical file name (Guardian file name)
PRIEXT	Integer	Primary extent size of file

**Table 5-6. ASSIGN Message Parts** (page 2 of 2)

Portion Name	Type	Identifies
SECEXT	Integer	Secondary extent size of file
FILECODE	Integer	File code
ACCESS	Integer	Access mode—input, output, or input/output
EXCLUSION	Integer	Exclusion mode—shared, exclusive, or protected
RECSIZE	Integer	Record size of file
BLKSIZE	Integer	Block size of file

## PARAM Message

The PARAM message contains all parameter names and the values associated with the names.

- A count of the number of named parameters.
- A list of the parameters in the following form:
  - Length of name
  - Name
  - Length of value
  - Value

## Using SMU Routines to Manipulate Messages

The following paragraphs summarize how you can manipulate messages using the SMU routines.

### Changing Environment Values

TACL ASSIGN, VOLUME, and PARAM commands let you specify default values that persist in your TACL environment until you explicitly change them or until you log off.

In the RUN command to start a process, you can specify IN and OUT files and a parameter string. These startup values override the default values for that process only and do not persist in your environment.

### Getting Environment Information

In the Guardian environment, a process can use the SMU functions to fetch the values of or store values into its PARAM, ASSIGN, and startup messages. Processes running in the OSS environment cannot use the SMU routines.

To access environment information using the SMU routines, your program must take the following steps:

- Save the messages.
- Use the Get family of SMU functions.

- Specify the message parts to retrieve.

[Table 5-7](#) on page 5-8 lists the message parts you can retrieve and the SMU function you use for each.

**Table 5-7. Retrievable Message Parts**

SMU Function	Message Part	Returns
SMU_Assign_GetText_	LOGICALNAME	Logical file name
	TANDEMNAME	Guardian file name
SMU_Assign_GetValue_	ACCESS	Access mode
	BLKSIZE	Block size
	EXCLUSION	Exclusion mode
	FILECODE	File code
	PRIEXT	Primary disk extent
	RECSIZE	Record size
	SECEXT	Secondary extent
SMU_Param_GetText_	The name of a parameter	The value of that parameter
SMU_Startup_GetText_	IN	Guardian file name of IN file
	OUT	Guardian file name of OUT file
	STRING	Value following command run-option
	VOLUME	Default volume and subvolume names

The Get subset of SMU functions enables your program to examine the names of files and their specified overrides before it tries to open the files. The program can thus be selective about the files it opens; it can choose among file descriptions according to the overrides it finds.

You can also use the Get subset of SMU functions to retrieve run-time parameters to a program. A program can take different actions based on the values specified in PARAM messages and command line information.

## Changing the Environment Information

You can use the Put subset of SMU functions to change the stored values and text in ASSIGN, PARAM, and startup messages. You can pass different environment information to descendant processes without changing the environment for subsequent RUN commands.

Each function in the Put subset is the exact counterpart of a function in the Get subset.

## Deleting Environment Information

You can use the Delete subset of SMU functions to delete entire ASSIGN, PARAM, and startup messages. You can also delete parts of ASSIGN and PARAM messages.

You cannot, however, delete parts of the startup message. Instead, you can use `SMU_Startup_PutText_` to assign null values to startup message parts, thereby achieving the same effect as deleting the parts.

## Using the environ Array

In the OSS environment, a process can access information in an array called *environ*. Such information is provided by the OSS file system.

In both the OSS and Guardian environments, C programs use the `getenv()` and `putenv()` functions to access the *environ* array. TAL and pTAL programs can use the `CRE_Getenv_` and `CRE_Putenv_` routines to access the *environ* array. COBOL saves messages only if you use the compiler SAVE directive. To access saved images of messages generated by TACL in the Guardian environment, use the Saved Message Utility (SMU).

In the Guardian environment, the CRE stores information in an array called *environ* if either of the following conditions are true:

- The value of the SAVE-ENVIRONMENT PARAM is ON when the process starts
- The process's main routine is written in C and the value of the SAVE-ENVIRONMENT PARAM is ON or the SAVE-ENVIRONMENT PARAM is not specified

If the value of the SAVE-ENVIRONMENT PARAM is OFF when the process starts, the CRE does not store environment information in the *environ* array.

To access information in the *environ* array from a program whose main routine is not written in C, you must enter the following TACL command before you run your program:

```
PARAM SAVE-ENVIRONMENT ON
```

The first four entries in the *environ* array are:

"STDIN"	Gives the file name of the standard input file, stdin.
"STDOUT"	Gives the file name of the standard output file, stdout.
"STDERR"	Gives the file name of the standard error file, stderr.
"DEFAULTS"	Gives the default volume and subvolume names used to qualify partial file names.

In addition to these four entries, the *environ* array contains the names and values of all parameters (PARAMs) in your environment when your process starts.





This section describes the interfaces to the CRE service functions. The service functions include:

- [Environment Functions](#) on page 6-1
- [File-Sharing Functions](#) on page 6-4
- [\\$RECEIVE Functions](#) on page 6-30
- [CRE\\_Terminator](#) on page 6-42
- [Exception-Handling Functions](#) on page 6-44

The TNS CRE service functions, data, and data structure declarations in TAL are available in the CREDECS file. The native CRE service functions, data, and data structure declarations in pTAL are available in the CRERDECS files.

C, COBOL, and FORTRAN run-time libraries call CRE service functions to access resources managed by the CRE. Do not call CRE service functions from C, COBOL, or FORTRAN routines; use each language's high-level constructs instead. Call CRE service functions only from TAL and pTAL routines that must share resources managed by the CRE with routines written in another language.

Many of the functions described in this section specify literal values that are predefined. For example, if you invoke `CRE_File_Close_`, you can use a predefined literal value to specify which standard file you are closing. The literal values used in this section are declared in the CREDECS file. See [Section 3, Compiling and Binding Programs for the TNS CRE](#), for information on using CREDECS.

## Environment Functions

This subsection describes the functions that retrieve and modify program environment variables. These functions are listed in [Table 6-1](#) on page 6-1.

**Table 6-1. CRE Environment Functions**

Function Name	Function Action
<a href="#">CRE_Getenv</a> on page 6-1	Returns a pointer to the value portion of an environment variable.
<a href="#">CRE_Putenv</a> on page 6-2	Stores a value in the value portion of an environment variable.

### CRE\_Getenv\_

The `CRE_Getenv_` function retrieves the address of the value portion of an environment variable.

The syntax for the TNS CRE environment is:

```
INT(32) PROC CRE_Getenv_( name );
    STRING .EXT name;                                ! in, required TNS only
```

*name*

is a pointer to a null-terminated string that specifies the name of the environment variable whose value `CRE_Getenv_` returns. *name* identifies a value established either in a `PARAM` command or by a previous call to `CRE_Putenv_`.

The syntax for the native CRE environment is:

```
EXTADDR PROC CRE_GETENV_="getenv"( Var_name ) LANGUAGE C;
  STRING .EXT name;
  EXTERNAL;                                ! in, required native only
```

*Var\_name*

is a pointer to a null-terminated string that specifies the name of the environment variable whose value `CRE_GETENV_` returns. *Var\_name* identifies a value established either in a `PARAM` command or by a previous call to `CRE_PUTENV_`.

## Return Value

`CRE_Getenv_` returns one of the following:

- The 32-bit address of the value of the environment variable specified in *name*
- Null (0D) if *name* is not the name of an environment variable

## Considerations

`CRE_Getenv_` corresponds to the C language `getenv( )` function. `CRE_Getenv_` is valid in large and wide memory model C programs running in the Guardian environment and in wide memory model C programs running in the OSS environment. `CRE_Getenv_` is not valid in small memory model programs. See the *C/C++ Programmer's Guide* for more information.

If `CRE_Getenv_` returns a nonzero value, the value it returns is the address of a null-terminated string.

Processes running in the Guardian environment can also use the Saved Message Utility (SMU) routines, described in [Section 9, Common Language Utility \(CLU\) Library Functions](#), to obtain environment information such as the process startup message, ASSIGNS, and PARAMs.

## CRE\_Putenv\_

The `CRE_Putenv_` function stores a value into an environment variable.

```
INT(32) PROC CRE_Putenv_( name-value );
  STRING .EXT name-value;      ! in, required  TNS only
```

*name-value*

is a null-terminated string that contains the name of an environment variable followed by an equal sign (=), followed by the value to store in the specified environment variable, as follows:

*name=value*

For example, the following string in *name-value*, stores the string “ABCDEFGH” in the environment variable “ALPHABET”:

ALPHABET=ABCDEFGH0

where the zero (0) is a null byte that terminates the *name-value* string.

The syntax for the native CRE environment is:

```
INT(32) PROC CRE_PUTENV_="putenv"( Var_string )LANGUAGE C;
  STRING .EXT Var_string;      ! in,   required
  EXTERNAL;                    !      native only
```

*Var\_string*

is a null-terminated string that contains the name of an environment variable followed by an equal sign (=), followed by the value to store in the specified environment variable, as follows:

*Var=string*

For example, the following string in *Var\_string*, stores the string “ABCDEFGH” in the environment variable “ALPHABET”:

ALPHABET=ABCDEFGH0

where the zero (0) is a null byte that terminates the *Var\_string* string.

## Return Value

CRE\_Putenv\_ returns one of the following:

- 0 if CRE\_Putenv\_ is successful
- A negative number, which is the negation of a CRE error number from the following table:

Error Code	Cause
31	Cannot obtain data space

## Considerations

CRE\_Putenv\_ corresponds to the C language `putenv( )` function.

If *name* (TNS CRE) exists, `CRE_Putenv_` stores *value* in the *name* environment variable. If *Var\_string* (native CRE) exists, `CRE_Putenv_` stores *string* in the *Var* environment variable.

If *name* (TNS CRE) does not exist, `CRE_Putenv_` creates a new environment variable called *name* and associates *value* with it. If *Var\_string* (native CRE) does not exist, `CRE_Putenv_` creates a new environment variable called *Var* and associates *string* with it.

`CRE_Putenv_` uses additional heap space, if needed.

`CRE_Putenv_` returns 0 unless it requires, but is unable to obtain, heap space in which case it returns error code 31. If it returns a nonzero value, *errno* is set to ENOMEM.

`CRE_Putenv_` is not valid in small memory model C programs. See the *C/C++ Programmer's Guide* for more information.

## File-Sharing Functions

The functions described in this subsection, which are listed in [Table 6-2](#) on page 6-4, support file sharing in the Guardian environment. The results of calls to these functions in the OSS environment are undefined. All of the functions listed are available in the native CRE library.

**Table 6-2. File-Sharing Functions**

Function Name	Function Action
<a href="#">CRE_File_Close</a> on page 6-5	Closes a standard file.
<a href="#">CRE_File_Control</a> on page 6-6	Sends an operating system control operation to a standard file.
<a href="#">CRE_File_Input</a> on page 6-8	Reads a record from standard input.
<a href="#">CRE_File_Message</a> on page 6-9	Sends a message to a standard file.
<a href="#">CRE_File_Open</a> on page 6-11	Opens a standard file.
<a href="#">CRE_File_Output</a> on page 6-20	Sends a record to standard output or standard log.
<a href="#">CRE_File_Retrycheck</a> on page 6-22	Determines whether an operation that resulted in an operating system error is retryable.
<a href="#">CRE_File_Setmode</a> on page 6-23	Sends an operating system SETMODE operation to a standard file.
<a href="#">CRE_Hometerm_Open</a> on page 6-24	Opens a process's home terminal.
<a href="#">CRE_Log_Message</a> on page 6-25	Writes a message to standard log.
<a href="#">CRE_Spool_Start</a> on page 6-27	Invokes the SPOOLSTART system procedure.

## CRE\_File\_Close\_

The CRE\_File\_Close\_ function closes the standard file that you specify.

The syntax for the TNS CRE environment is:

```
INT PROC CRE_File_Close_( file_ordinal, disposition, cplist )
EXTENSIBLE;
  INT      file_ordinal;      ! in,      required
  INT      disposition;      ! in,      optional
  INT(32) .cplist;           ! in/out, optional  TNS only
```

The syntax for the native CRE environment is:

```
INT PROC CRE_File_Close_( file_ordinal, disposition, cplist )
EXTENSIBLE;
  INT      file_ordinal;      ! in,      required
  INT      disposition;      ! in,      optional
  INT(32) .EXT cplist;       ! in/out, optional  native only
```

*file\_ordinal*

identifies the standard file to close. You can use the following symbolic names for *file\_ordinal*:

### CRE File Ordinal Symbolic Names

CRE^Standard^Input  
CRE^Standard^Output  
CRE^Standard^Log

*disposition*

is an optional parameter that specifies the tape position parameter for a call to the FILE\_CLOSE\_ system procedure.

*cplist*

if present and its address value is nonzero in the TNS environment, is a checkpoint list. In the native CRE environment, this parameter has no meaning and should be left empty (it is used by COBOL for process pair execution).

## Return Value

CRE\_File\_Close\_ returns:

- 0 if the close operation is successful
- A positive number, which is a file system error number

- A negative number, which is the negation of a CRE error number from the following table:

Error Code	Cause
16	Checkpoint list exhausted
55	Missing or invalid parameter
63	Undefined shared file

## Considerations

CRE\_File\_Close\_ decrements its count of the number of connections it has granted. If after decrementing its count by one, the number of outstanding connections is nonzero, CRE\_File\_Close\_ returns to its caller without taking further action. If the result of decrementing the number of outstanding connections goes to zero, CRE\_File\_Close\_ calls:

- The CLOSEEDIT system procedure if the file was opened by a call to OPENEDIT.
- The SPOOLEND system procedure and then the FILE\_CLOSE\_ system procedure if the file was a spooler collector that was open for buffered (level-3) spooling.
- The FILE\_CLOSE\_ system procedure for all other files.

## CRE\_File\_Control\_

The CRE\_File\_Control\_ function invokes the CONTROL or SPOOLCONTROL system procedures, passing the parameters you specify when you call CRE\_File\_Control\_.

```
INT PROC CRE_File_Control_( file_ordinal, operation, param );
  INT file_ordinal;           ! in,  required
  INT operation;             ! in,  required
  INT param;                 ! in,  required   TNS,native
```

*file\_ordinal*

identifies the standard file to which to send a control operation. You can use the following symbolic names for *file\_ordinal*:

```
CRE^Standard^Input
CRE^Standard^Output
CRE^Standard^Log
```

*operation*

is the operation parameter to pass to the CONTROL or SPOOLCONTROL system procedures.

*param*

is the *param* parameter to pass to the CONTROL or SPOOLCONTROL system procedures.

## Return Value

CRE\_File\_Control\_ returns one of the following:

- 0 if the control operation is successful
- A positive number, which is a file system error number
- A negative number, which is the negation of a CRE error number from the following table:

Error Code	Meaning
55	Missing or invalid parameter
57	Parameter value not accepted
63	Undefined shared file
64	File not open

## Considerations

If the standard file associated with *file\_ordinal* is a spooler collector, CRE\_File\_Control\_:

- Invokes the SPOOLSTART system procedure, if SPOOLSTART has not been invoked yet for *file\_ordinal*.
- Invokes the SPOOLCONTROL system procedure, passing *operation* and *param* to the corresponding SPOOLCONTROL parameters.

If the standard file associated with *file\_ordinal* is not a spooler collector, CRE\_File\_Control\_ calls the CONTROL system procedure, passing *operation* and *param* to the CONTROL procedure.

CRE\_File\_Control\_ does not retry operations that return an error.

For more information on the SPOOLSTART and SPOOLCONTROL system procedures, see the *Spooler Programmer's Guide*.

For more information on the CONTROL system procedure, see the *Guardian Procedure Calls Reference Manual*.

## CRE\_File\_Input\_

The CRE\_File\_Input\_ function reads a record from a standard file.

```

INT PROC CRE_File_Input_( file_ordinal, buffer:read_count,
                          count_read, write_count )
                          EXTENSIBLE;

  INT      file_ordinal;      ! in,  required
  STRING .EXT buffer;         ! out, required
  INT      read_count;        ! in,  required
  INT .EXT count_read;        ! out, optional
  INT      write_count;       ! in,  optional      TNS,native

```

*file\_ordinal*

identifies the standard file from which to read. You can use only the symbolic name CRE^Standard^Input for *file\_ordinal*.

*buffer:read\_count*

defines the data area (*buffer*) in which to store the data read. *read\_count* specifies the maximum number of bytes to read.

*count\_read*

if present, is the number of bytes read into *buffer* when CRE\_File\_Input\_ returns control to your program. If no bytes are read, *count\_read* is zero.

*write\_count*

if present and has a value other than -1, is the number of bytes in *buffer* to write as a prompt before reading from the file. If you do not specify *write\_count*, a prompt is not issued.

## Return Value

CRE\_File\_Input\_ returns one of the following:

- 0 if the input operation is successful
- A positive number, which is a file system error number
- A negative number, which is the negation of a CRE error number from the following table:

Error Code	Cause
55	Missing or invalid parameter
56	Invalid parameter value
63	Undefined shared file
64	File not open
78	EDITREAD failed



## Considerations

CRE\_File\_Input\_ reads the next sequential record from the file referenced by *file\_ordinal* unless *file\_ordinal* specifies a null file in which case CRE\_File\_Input\_ returns end of file without taking any other action.

If *file\_ordinal* does not reference a null file, CRE\_File\_Input\_ reads the next sequential record by calling:

- READEDIT if the file is an EDIT file that was opened by calling OPENEDIT.
- EDITREAD if the file is an EDIT file that was opened by calling FILE\_OPEN\_ and EDITREADINIT.
- WRITEREADX if the file is a terminal or process (other than a spooler collector) and *write\_count* is not equal to -1. In this case, *write\_count* specifies the number of bytes to write from *buffer* during the write portion of the WRITEREADX operation.
- READX for all other cases.

CRE\_File\_Input\_ does not retry operations that return an error.

## CRE\_File\_Message\_

The CRE\_File\_Message\_ function sends a message to a standard file.

```

INT PROC CRE_File_Message_( file_ordinal,
                             buffer:message_bytes,
                             indent_bytes, read_count,
                             count_read )
    EXTENSIBLE;

    INT      file_ordinal;      ! in,      required
    STRING .EXT buffer;         ! in/out,  required
    INT      message_bytes;    ! in,      required
    INT      indent_bytes;     ! in,      optional
    INT      read_count;       ! in,      optional
    INT      .EXT count_read;   ! out,     optional TNS,native

```

*file\_ordinal*

identifies the standard file to which to send a message. You can use the following symbolic names for *file\_ordinal*:

### CRE File Ordinal Symbolic Names

CRE^Standard^Output

CRE^Standard^Log

*buffer:message\_bytes*

is the message to send. If *read\_count* is not -1, CRE\_File\_Message\_ stores a response, if one is received, at *buffer*.

*indent\_bytes*

if present, specifies whether or not CRE\_File\_Message\_ should write your message on multiple lines, if necessary and, if so, the indentation to use for all lines after the first.

CRE\_File\_Message\_ alters the contents of *buffer* as follows, if your message requires more than one line:

- If *indent\_bytes* > 0, each message line after the first line is indented by *indent\_bytes* characters.
- If *indent\_bytes* = 0, message lines after the first line are not indented.
- If *indent\_bytes* = -1, the message is written in its entirety on one line. CRE\_File\_Message\_ returns an error if the length of the message exceeds the number of bytes that can be written to the output device.
- If *indent\_bytes* < -1, each message line after the first line uses the first *absolute\_value(indent\_bytes)* characters of the first line of the message as a prefix.

*read\_count*

if present, is the maximum number of bytes CRE\_File\_Message\_ reads in response to the message it writes. CRE\_File\_Message\_ returns a response only if *read\_count* is greater than or equal to zero and the device type associated with *file\_ordinal* is a terminal or a process (other than a spooler collector). The response is stored in *buffer*. If you do not specify *read\_count*, CRE\_File\_Message\_ does not return a response.

*count\_read*

if present, is the actual number of bytes in the response. CRE\_File\_Message\_ sets *count\_read* to -1 if it does not attempt to read a response or if its attempt to read a response is not successful.

## Return Value

CRE\_File\_Message\_ returns one of the following:

- 0 if the message operation is successful
- A positive number, which is a file system error number
- A negative number, which is the negation of a CRE error number from the following table:

Error Code	Cause
55	Missing or invalid parameter
56	Invalid parameter value

Error Code	Cause (continued)
57	Parameter value not accepted
63	Undefined shared file
64	File not open

## Considerations

CRE\_File\_Message\_ retries operations that return an error and are recoverable.

## CRE\_File\_Open\_

The CRE\_File\_Open\_ function opens a standard file.

The syntax for the TNS CRE environment is:

```

INT PROC CRE_File_Open_( file_ordinal, flags, access,
                        exclusion, no_wait,
                        sync_receive_depth, options,
                        cplist ) EXTENSIBLE;

INT      file_ordinal;      ! in,      required
INT      flags;              ! in,      optional
INT      access;             ! in,      optional
INT      exclusion;           ! in,      optional
INT      no_wait;            ! in,      optional
INT      sync_receive_depth; ! in,      optional
INT      options;            ! in,      optional
INT(32)  .cplist;            ! in/out, optional  TNS only

```

The syntax for the native CRE environment is:

```

INT PROC CRE_File_Open_( file_ordinal, flags, access,
                        exclusion, no_wait,
                        sync^receive_depth, options,
                        cplist ) EXTENSIBLE;

INT      file_ordinal;      ! in,      required
INT      flags;              ! in,      optional
INT      access;             ! in,      optional
INT      exclusion;           ! in,      optional
INT      no_wait;            ! in,      optional
INT      sync_receive_depth; ! in,      optional
INT      options;            ! in,      optional
INT(32)  .EXT cplist;        ! in/out, optional native only

```

*file\_ordinal*

identifies the standard file to open. You can use the following symbolic names for *file\_ordinal*:

**CRE File Ordinal Symbolic Names**


---

CRE^Standard^Input

CRE^Standard^Output

CRE^Standard^Log

*flags*

if present, specifies CRE options. *flags*.<0:13> must be 0. *flags*.<14:15> specifies the buffering attribute for a spooler collector. If *flags* is not passed, CRE\_File\_Open\_ assumes its value to be 0.

*access*

if present, specifies read-only, write-only, or read-write access to the file. If the file is not open and you either do not specify *access* or specify its value as -1, CRE\_File\_Open\_ uses:

- Read-only access if you are opening standard input
- Write-only access if you are opening standard output or standard log

If the file is already open and you do not specify *access* or specify its value as -1, CRE\_File\_Open\_ sets the access for the current open to the same value used when the file was first opened.

CRE\_File\_Open\_ converts extend access to write-only access.

*access* corresponds to the access parameter to the FILE\_OPEN\_ system procedure.

Refer to the FILE\_OPEN\_ procedure in the *Guardian Procedure Calls Reference Manual* for valid access parameter values.

*exclusion*

if present, specifies shared, exclusive, or protected access to the file. If the file is not open and you either do not specify *exclusion* or specify its value as -1, CRE\_File\_Open\_ uses:

- Shared if the device is a terminal
- Protected if the device is a process or a disk and the access mode is read-only
- Exclusive in all other cases

If the file is already open and you do not specify *exclusion* or you specify its value as -1, CRE\_File\_Open\_ sets *exclusion* for the current open to the value used when the file was first opened.

*exclusion* corresponds to the exclusion parameter to the FILE\_OPEN\_ system procedure.

Refer to the FILE\_OPEN\_ procedure in the *Guardian Procedure Calls Reference Manual* for valid access parameter values.

*no\_wait*

if present, is the nowait parameter to pass to the FILE\_OPEN\_ system procedure. CRE\_File\_Open\_ returns an error if you specify a nonzero value for *no\_wait*. If you do not specify *no\_wait*, CRE\_File\_Open\_ uses a value of zero.

*sync\_receive\_depth*

if present, is the sync-or-receive-depth parameter to pass to the FILE\_OPEN\_ system procedure. The value passed to FILE\_OPEN\_ is determined as follows:

- If *sync\_receive\_depth* is present and nonzero, CRE\_File\_Open\_ passes *sync\_receive\_depth* to FILE\_OPEN\_.
- If *sync\_receive\_depth* is present and zero, CRE\_File\_Open\_ passes zero to FILE\_OPEN\_ unless:
  - Your program is compiled to run as a process pair
  - Your program includes more than one language (not including TAL and pTAL)

In these cases, the CRE specifies 1 for the FILE\_OPEN\_ sync-or-receive-depth parameter.

- If *sync\_receive\_depth* is not present, the CRE specifies 1 for the FILE\_OPEN\_ sync-or-receive-depth parameter, except for \$RECEIVE, for which it specifies 0.

*options*

if present, is the options parameter for FILE\_OPEN\_. If you do not specify *options*, CRE\_File\_Open\_ passes zero to FILE\_OPEN\_, unless the access mode is read-only and the file is \$RECEIVE (TNS CRE environment only), in which case it passes one to FILE\_OPEN\_.

*cplist*

if present and its address value is nonzero in the TNS environment, is a checkpoint list. In the native CRE environment, this parameter has no meaning and should be left empty (it is used by COBOL for process pair execution).

## Return Value

CRE\_File\_Open\_ returns one of the following:

- 0 if the open operation is successful

- A positive number, which is a file system error number
- A negative number, which is the negation of one of the following CRE error numbers:

Error Code	Cause
16	Checkpoint list exhausted
17	Cannot obtain control space
20	Cannot utilize file name
55	Missing or invalid parameter
57	Parameter value not accepted
63	Undefined shared file
65	Invalid attribute value
66	Unsupported file device
67	Access mode not accepted
68	Nowait value not accepted
69	Syncdepth not accepted
71	Inconsistent attribute value
77	EDITREADINIT failed
79	OPENEDIT failed

## Considerations

- Opening standard files

The CRE manages shared access to only three files: standard input, standard output, and standard log. Standard files are used as a source of sequential records when specified for input (standard input) and as a destination to which to write sequential records when specified as output (standard output and standard log). See [Section 2, CRE Services](#), for a full description of the three standard files.

To open a standard file, your program must call `CRE_File_Open_`. Except for routines written in TAL, the run-time library for each language calls `CRE_File_Open_`. Only TAL routines must call `CRE_File_Open_` explicitly.

For each standard file, your program must call `CRE_File_Open_` before it calls any other CRE functions for that file. `CRE_File_Open_` opens each standard file according to the I/O model of the language in which your program's main routine is written.

To access files other than standard files, your program (or its run-time environment) must use standard file system procedures such as the `FILE_OPEN_` system procedure.

The device type for standard input must be a process, `$RECEIVE` (in the TNS CRE environment), a disk, or a terminal.

The device type for standard output and standard log must be a process, the operator console, a disk, a printer, or a terminal.

If you specify different values for the *exclusion*, *sync\_receive\_depth*, or *options* parameters on any open after the first open, the CRE returns an error stating that the requested attributes are incompatible with existing attributes, unless the requested attributes are a subset of the existing attributes.

- File connections

For each standard file, the CRE opens a file one time, regardless of how many times your program opens that standard file. The CRE grants a connection to the file to each caller of `CRE_File_Open_` that specifies the same standard file—standard input, standard output, or standard log. Because `CRE_File_Open_` grants a connection to the same open of the file to each caller, each reference to the file reads or writes, as appropriate, to the same file.

Under some circumstances, the CRE does not open a physical file. The CRE does, however, grant connections to the standard file, even though it does not open a file.

[Figure 6-1](#) on page 6-16 shows how the CRE manages connections to standard files. As shown in the figure, `CRE_File_Open_` searches its I/O file table for an entry that corresponds to the standard file that you are opening. If an entry exists, `CRE_File_Open_` grants the caller a connection to the file. Otherwise, `CRE_File_Open_` calls the `FILE_OPEN_` system procedure to open the file.

See [Section 2, CRE Services](#), for a full description of how the CRE determines the name of the file to open for each standard file.

- Accessing standard files

Use the standard constructs of the languages in which you write your routines to access standard files. [Table 6-3](#) on page 6-16 shows examples of how your program accesses the standard files supported by the CRE. For more information, see the reference manuals for the languages in which your routines are written.

Figure 6-1. Using Connections to Share a File Open

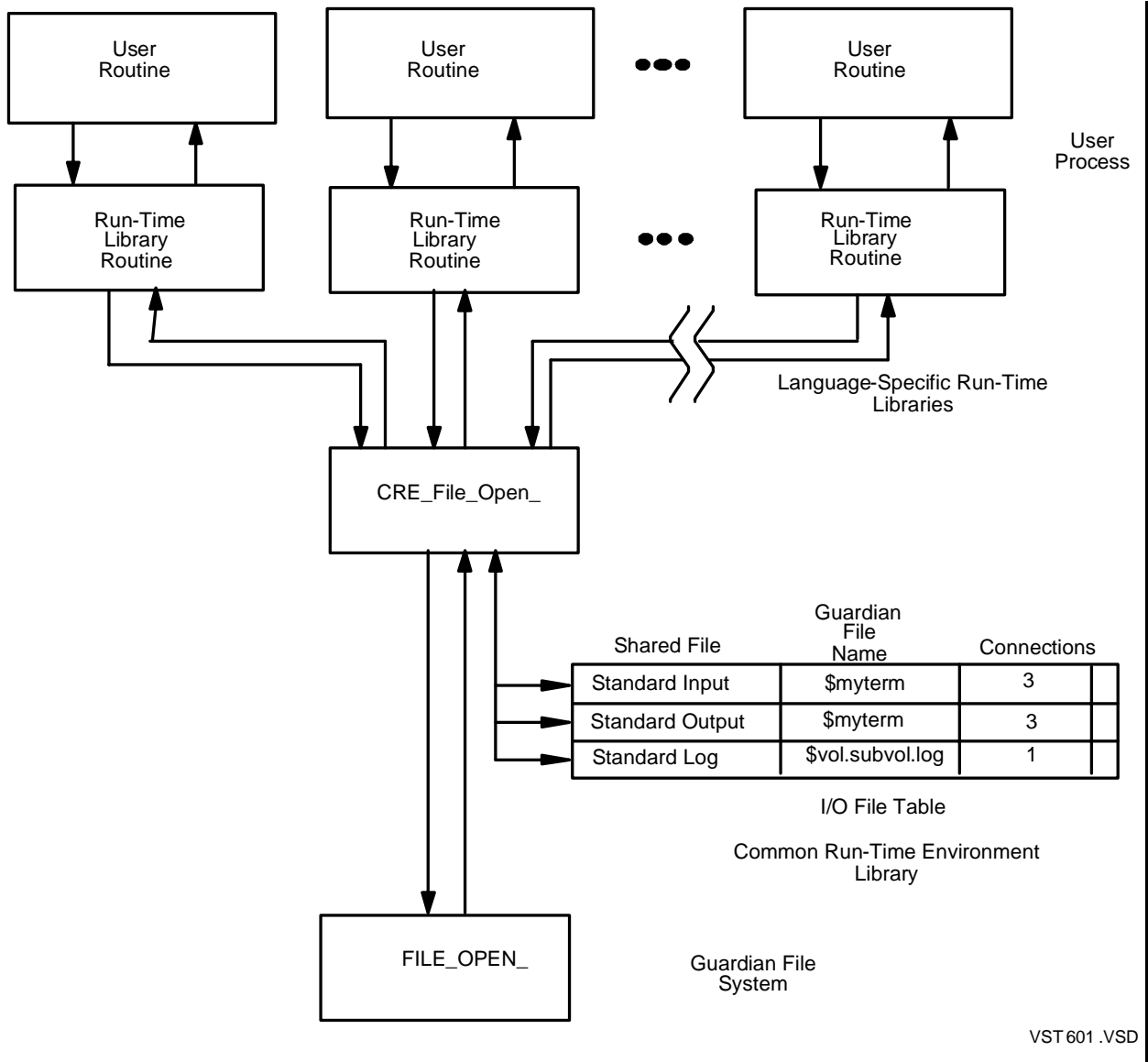


Table 6-3. Language-Specific Constructs for Standard Files

Language	Standard File		
	Standard Input	Standard Output	Standard Log
C	str = gets(sp); ch = getchar(); scanf("%s", sp);	puts(sp); putchar(ch); printf("hello",\n);	perror(sp);
COBOL	ACCEPT sp	DISPLAY sp	Not available
FORTRAN	READ(5,100) sp	WRITE(6,100) sp	PAUSE string STOP string



- Opening terminals and processes

A standard file can access a terminal or a process (except a spooler collector) for both input and output. If the file you specify for standard input is the same as the file you specify for standard output or standard log, the CRE opens the operating system file only once, although you must invoke `CRE_File_Open_` once for the read component of the open, and once for the write component of the open. The CRE, however, opens the operating system file only once and specifies read-write access.

- Disk files

You can specify an existing or nonexistent disk file as a standard file. If you specify a file that does not exist, `CRE_File_Open_` attempts to create a file. If it is unable to create a file, either because you specified an unacceptable access (for example, you open a nonexistent file as read-only) or because the operating system returned an error when `CRE_File_Open_` attempted to create the file, `CRE_File_Open_` returns an error.

- The `FILE_OPEN_` and `EDITREADINIT` system procedures if your program's main routine is written in COBOL or FORTRAN, specifies that it is to run as a process pair, and opens the file for read-only access. If your FORTRAN or COBOL routine specifies that it is to run as a process pair and attempts to open an EDIT file with an access mode other than read-only, `CRE_File_Open_` returns error 67, "access mode not accepted."
- Files other than EDIT files  
`CRE_File_Open_` opens all files other than EDIT files by calling the `FILE_OPEN_` system procedure.

---

**Note.** Some of the descriptions in this subsection specify different CRE actions depending on whether a program's main routine specifies that the program runs as a process pair. In all such cases, the actions of the CRE are based solely on whether the main routine specifies that the program runs as a process pair, regardless of whether the program initiates process pairs or whether the program is running as a process pair.

---

- EDIT files  
`CRE_File_Open_` opens EDIT files using:
  - The `OPENEDIT_` system procedure if your program's main routine is written in C/C++, pTAL, or TAL, or is written in COBOL or FORTRAN and does not specify that it is to run as a process pair.
- Nonexistent disk files  
If your program specifies a nonexistent disk file and an access mode other than read-only, `CRE_File_Open_` attempts to create a file for you. `CRE_File_Open_` creates:

- An EDIT file if your program's main routine is written in C/C++, pTAL, or TAL, or if your program's main routine is written in COBOL or FORTRAN and does not specify that it is to run as a process pair.
  - An entry-sequenced file if your program's main routine is written in COBOL or FORTRAN and specifies that it is to run as a process pair (regardless of whether your program is currently running as a process pair).
- Spooler collectors

You can specify a spooler collector as the external file for either standard output or standard log.

Although you specify a spooler collector, the CRE does not call the SPOOLSTART system procedure during CRE\_File\_Open\_. Your program can call CRE\_Spool\_Start\_ to establish parameters to the spooler collector (for example, the number of copies you want printed, or the location at which to print your listing) after you call CRE\_File\_Open\_ but before you initiate any other CRE I/O function.

If your program specifies a spooler collector for output but does not call CRE\_Spool\_Start\_ before its first call to a CRE I/O function, the CRE calls SPOOLSTART if it determines that it needs to do so (for example, if you have specified buffered spooling but have not called CRE\_Spool\_Start\_ or if you specified CRE^Undecided^spooling).

If the file referenced by *file\_ordinal* is a spooler collector, the *flags* parameter specifies the type of spooling you want. You can use the following literals when you call CRE\_File\_Open\_. (The literals are declared in the CREDECS file.)

Spooling Type	Meaning
CRE^Simple^spooling	Unbuffered spooling (level-1 or level-2 spooling)
CRE^Undecided^spooling	Spooling type determined by the CRE
CRE^Buffered^spooling	Buffered spooling (level-3 spooling)

The following example shows a call to CRE\_File\_Open\_ that specifies a spooling literal.

```
CALL CRE_File_Open_(file_no, CRE^Buffered^spooling,... );
```

- Simple Spooling

If you specify CRE^Simple^spooling, the CRE does not allocate a buffer for spooling and writes output directly to the spooler collector by calling the WRITE system procedure. Your program uses level-1 spooling unless it calls CRE\_Spool\_Start\_, in which case it uses level-2 spooling.

- Buffered Spooling

If you specify CRE^Buffered^spooling, the call to CRE\_File\_Open\_ fails if:

- *file\_ordinal* specifies standard log.

- The CRE is unable to allocate a buffer for spooled output.
- Undecided Spooling

If you specify CRE^Undecided^spooling, pass zeroes for *flags*, or do not specify *flags*, the CRE converts your request to CRE^Simple^spooling if:

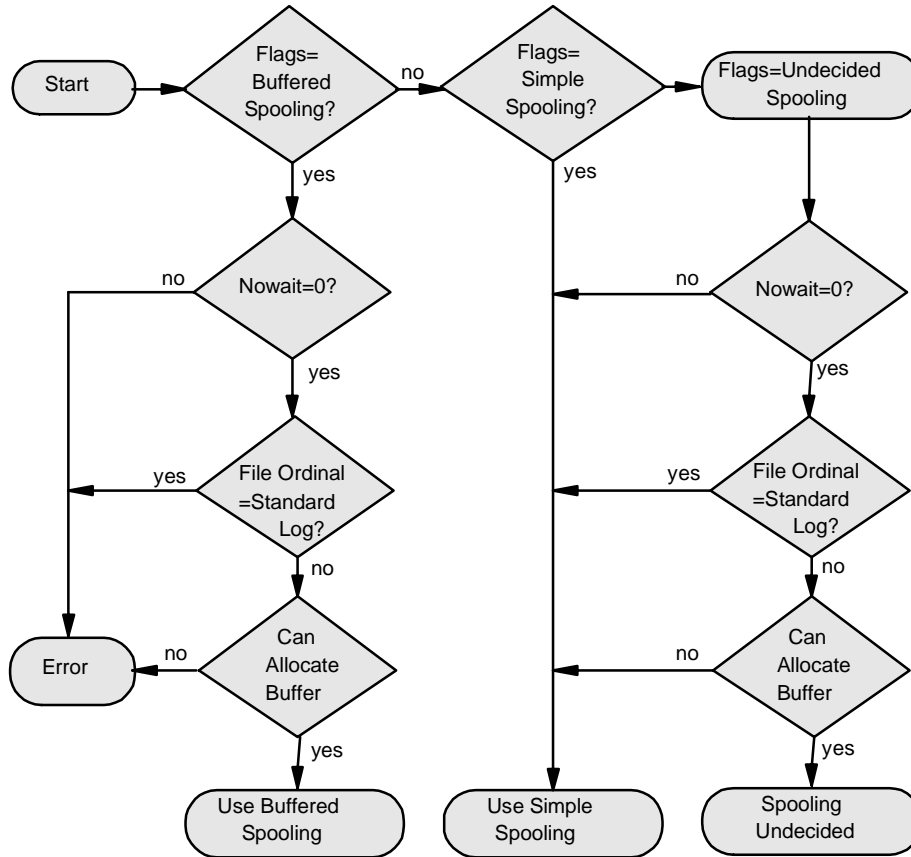
  - *file\_ordinal* specifies standard log.
  - The CRE is unable to allocate a buffer for spooled output.

If you specify CRE^Undecided^spooling when you open the file, and the CRE does not convert your request to CRE^Simple^spooling as described in the preceding text, the CRE determines whether to use simple or buffered spooling as follows:

The flow chart in [Figure 6-2](#) on page 6-20 shows how CRE\_File\_Open\_ processes your open request when you specify a spooler collector.

For more information on the buffered spooling, see [CRE\\_Spool\\_Start](#) on page 6-27.

For more information about spooling, refer to the *Spooler Utilities Reference Manual* and the *Spooler Programmer's Guide*.

**Figure 6-2. Determining Spooler Buffering in CRE\_File\_Open\_**

VST 602 .VSD

## CRE\_File\_Output\_

The CRE\_File\_Output\_ function writes a record to a standard file.

```

INT PROC CRE_File_Output_( file_ordinal, buffer:write_count,
                           count_written, spacing_option )
    EXTENSIBLE;
    INT      file_ordinal;      ! in,  required
    STRING .EXT buffer;        ! in,  required
    INT      write_count;      ! in,  required
    INT      .EXT count_written; ! out, optional
    INT      spacing_option;    ! in,  optional    TNS,native
  
```

*file\_ordinal*

identifies the standard file to which to write. You can use the following symbolic names for *file\_ordinal*:

#### **CRE File Ordinal Symbolic Names**

---

CRE^Standard^Output

CRE^Standard^Log

*buffer:write\_count*

is the record to transmit (*buffer*) and its length in bytes (*write\_count*).

*count\_written*

if present, is the number of bytes written when the record is transmitted. If a record is not transmitted, *count\_written* is zero. See the following considerations for more information on *count\_written*.

*spacing\_option*

if present, specifies spacing actions. If you do not pass *spacing\_option*, CRE\_File\_Output\_ uses a value of one.

*spacing\_option* is applicable when the target device of CRE\_File\_Output\_ is a process, line printer, spooler collector, or terminal. The bits in *spacing\_option* are defined as follows:

<b>Bits in <i>spacing_option</i></b>	<b>Meaning</b>
0	On this operation: 0: Write before advancing 1: Write after advancing
1	Default spacing mode: 0: Write before advancing 1: Write after advancing
2:3	Reserved
4:15	Number of lines to advance

You can use *spacing\_option*.<4:15>, number of lines to advance, to obtain particular effects. For example, a series of lines written with *spacing\_option* 2 creates double-spaced lines. A series of lines written with *spacing\_option* 0 creates overprinted lines.

## **Return Value**

CRE\_File\_Output\_ returns one of the following:

- 0 if the output operation is successful
- A positive number, which is a file system error number

- A negative number, which is the negation of a CRE error number from the following table:

Error Code	Cause
55	Missing or invalid parameter
57	Parameter value not accepted
63	Undefined shared file
64	File not open

## Considerations

- If the file referenced by *file\_ordinal* is a spooler collector, the current spooling state is either buffered spooling or undecided spooling, and the spooler has not been initialized, CRE\_File\_Output\_ calls the CRE\_Spool\_Start\_ procedure.
- If the file referenced by *file\_ordinal* is a spooler collector open for level-3 spooling, CRE\_File\_Output\_ calls SPOOLWRITE.
- If *file\_ordinal* specifies a null file—that is, CRE\_File\_Open\_ has been called and has granted one or more connections to the file but an operating system file was not opened— CRE\_File\_Output\_ sets *count\_written* to the value specified in *write\_count*, and returns 0 as the value of the function.
- If the file was opened by a call to the OPENEDIT system procedure, CRE\_File\_Output\_ calls the WRITEEDIT system procedure.
- In all other cases, CRE\_File\_Output\_ calls the WRITEX system procedure to write the contents of *buffer*.
- CRE\_File\_Output\_ does not retry operations that return an error.

## CRE\_File\_Retrycheck\_

The CRE\_File\_Retrycheck\_ function determines whether an I/O operation that completed with an operating system error should be retried.

```
INT PROC CRE_File_Retrycheck_( file_ordinal, error )
                                EXTENSIBLE;
    INT file_ordinal;           ! in,  required
    INT error;                  ! in,  optional   TNS,native
```

*file\_ordinal*

identifies the standard file for which to check for a retryable operation. You can use the following symbolic names for *file\_ordinal*:

```
CRE^Standard^Input
CRE^Standard^Output
CRE^Standard^Log
```

*error*

if present and greater than or equal to zero, is a file system error code; otherwise, if present and less than zero, *error* is a negated CRE error ordinal.

## Return Value

CRE\_File\_Retrycheck\_ returns 0 if *file\_ordinal* is not valid or if *error* is present and has a negative value; otherwise, CRE\_File\_Retrycheck\_ returns the value returned to it from a call to the FILEERROR system procedure. See “FILEERROR” in the *Guardian Procedure Calls Reference Manual* for more information on the FILEERROR procedure.

## Considerations

- Retrying an operation is appropriate only if:
  - The last operation failed.
  - The failure was detected by the operating system.
  - The cause of the failure is considered recoverable.
- CRE\_File\_Retrycheck\_ does not return errors.

## CRE\_File\_Setmode\_

The CRE\_File\_Setmode\_ function invokes the SETMODE or the SPOOLSETMODE system procedure, passing the parameters you specify when you call CRE\_File\_Setmode\_.

```
INT PROC CRE_File_Setmode_( file_ordinal, function, param1,
                           param2 ) EXTENSIBLE;
  INT file_ordinal;           ! in,  required
  INT function;              ! in,  required
  INT param1;                ! in,  required
  INT param2;                ! in,  optional   TNS,native
```

*file\_ordinal*

identifies the standard file to which to send a setmode operation. You can use the following symbolic names for *file\_ordinal*:

```
CRE^Standard^Input
CRE^Standard^Output
CRE^Standard^Log
```

*function*

is the function parameter for a SETMODE or SPOOLSETMODE system procedure.

*param1*

is the *param1* parameter for a SETMODE or SPOOLSETMODE system procedure.

*param2*

is the *param2* parameter for a SETMODE or SPOOLSETMODE system procedure.

## Return Value

CRE\_File\_Setmode\_ returns one of the following:

- 0 if the setmode operation is successful
- A positive number, which is a file system error number
- A negative number, which is the negation of a CRE error number from the following table:

Error Code	Cause
55	Missing or invalid parameter
63	Undefined shared file
64	File not open

## Considerations

- If the standard file associated with *file\_ordinal* specifies buffered spooling or undecided spooling and CRE\_Spool\_Start\_ has not been invoked, CRE\_File\_Setmode\_ calls CRE\_Spool\_Start\_.
- If the file specified by *file\_ordinal* is a spooler collector and your program is using level-3 spooling for that file, CRE\_File\_Setmode\_ invokes the SPOOLSETMODE system procedure, passing the parameters to CRE\_File\_Setmode\_ to SPOOLSETMODE. Otherwise, CRE\_File\_Setmode\_ invokes the SETMODE system procedure passing the parameters to CRE\_File\_Setmode\_ to SETMODE.
- CRE\_File\_Setmode\_ does not retry operations that return an error.

## CRE\_Hometerm\_Open\_

The CRE\_Hometerm\_Open\_ function opens your process's home terminal.

```
INT PROC CRE_Hometerm_Open_( file_number );
    INT .EXT file_number;          ! out          TNS,native
```

*file\_number*

is assigned the file number of your process's home terminal if the open operation succeeds; otherwise, CRE\_Hometerm\_Open\_ assigns -1 to *file\_number*.



## Return Value

CRE\_Hometerm\_Open\_ returns one of the following:

- 0 if the open operation is successful
- A positive number, which is a file system error number
- A negative number, which is the negation of a CRE error number from the following table:

Error Code	Cause
20	Cannot utilize file name

## Considerations

- CRE\_Hometerm\_Open\_ opens your process's home terminal and returns its file number in *file\_number*.
- If the CRE receives an EXECUTION-LOG PARAM at process initialization, CRE\_Hometerm\_Open\_ does not store a value in *file\_number* and returns error 20, "Cannot utilize file name."
- CRE\_Hometerm\_Open\_ does not retry operations that return an error.

## CRE\_Log\_Message\_

The CRE\_Log\_Message\_ function writes a message to standard log. Although you should use standard constructs in your programming language, you can directly invoke CRE\_Log\_Message\_ from your application program to log a message.

```

INT PROC CRE_Log_Message_( buffer:message_bytes,
                           indent_bytes,
                           read_count, count_read )
                           EXTENSIBLE;

  STRING .EXT buffer;           ! in/out, required
  INT     message_bytes;        ! in,      required
  INT     indent_bytes;         ! in,      optional
  INT     read_count;           ! in,      optional
  INT     .EXT count_read;      ! out,    optional TNS, native

```

*buffer:message\_bytes*

defines the address and length of the message to transmit.

*indent\_bytes*

if present, specifies formatting of the message.

*read\_count*

if present, specifies the number of bytes to read in response to the message.

*count\_read*

if present, is the number of bytes in the response.

For details on the parameters to `CRE_Log_Message_`, see the descriptions of the corresponding parameters in the `CRE_File_Message_` procedure.

## Return Value

`CRE_Log_Message_` returns one of the following:

- 0 if the message operation is successful
- A positive number, which is a file system error number
- A negative number, which is the negation of a CRE error number from the following table:

Error Code	Cause
17	Cannot obtain control space
20	Cannot utilize file name
55	Missing or invalid parameter
56	Invalid parameter value
57	Parameter value not accepted
66	Unsupported file device
71	Inconsistent attribute value
79	OpenEdit failed

## Considerations

- `CRE_Log_Message_` sends a message to standard log. If standard log is not open, `CRE_Log_Message_` opens it.
- `CRE_Log_Message_` retries operations that cause recoverable errors.

## CRE\_Spool\_Start\_

The CRE\_Spool\_Start\_ function invokes the SPOOLSTART system procedure.

```

INT PROC CRE_Spool_Start_(file_ordinal, buffering,
                           location:loc_bytes,
                           form_name:form_bytes,
                           report_name:report_bytes,
                           number_of_copies, page_size,
                           flags, owner, max_lines,max_pages)
                           EXTENSIBLE;

INT          file_ordinal;      ! in,  required
INT          buffering;         ! in,  optional
STRING .EXT  location;          ! in,  optional
INT          loc_bytes;         ! in,  optional
STRING .EXT  form_name;         ! in,  optional
INT          form_bytes;        ! in,  optional
STRING .EXT  report_name;       ! in,  optional
INT          report_bytes;      ! in,  optional
INT          number_of_copies;  ! in,  optional
INT          page_size;         ! in,  optional
INT          flags;             ! in,  optional
INT          owner;             ! in,  optional
INT(32)      max_lines;         ! in,  optional
INT(32)      max_pages;         ! in,  optional   TNS, native

```

### *file\_ordinal*

identifies the standard file for which to initiate SPOOLSTART. You can use the following symbolic names for *file\_ordinal*:

CRE^Standard^Output  
CRE^Standard^Log

### *buffering*

if present, specifies the buffering attribute for the spooler job; if *buffering* is not passed, CRE\_Spool\_Start\_ uses 0. You can use the symbolic names in the following table to specify *buffering*:

CRE^Simple^spooling  
CRE^Buffered^spooling  
CRE^Undecided^spooling

The preceding literals are defined in the CREDECS file for the TNS CRE, and the CRERDECS file for the TNS/R or TNS/E native CRE. Refer to [Section 3, Compiling and Binding Programs for the TNS CRE](#), for information on the CREDECS file. Refer to [Section 4, Compiling and Linking Programs for the Native CRE](#), for information on the CRERDECS file.

[Figure 6-3](#) on page 6-30 shows the algorithm CRE\_Spool\_Start\_ uses to choose simple spooling or buffered spooling for the file associated with *file\_ordinal*.

*location:location\_bytes*

if present, specifies the location for the spooler job and its length in bytes. CRE\_Spool\_Start\_ passes this parameter to the SPOOLSTART system procedure's location parameter. CRE\_Spool\_Start\_ ensures that the string it passes to SPOOLSTART is 16 bytes long by truncating or adding blank characters to the string you specify.

*form\_name:form\_bytes*

if present, specifies the form name for the spooler job. CRE\_Spool\_Start\_ passes this parameter to the SPOOLSTART system procedure's form-name parameter. CRE\_Spool\_Start\_ ensures that the string it passes to SPOOLSTART is 16 bytes long by truncating or adding blank characters to the string you specify.

*report\_name:report\_bytes*

if present, specifies the report name for the spooler job. CRE\_Spool\_Start\_ passes this parameter to the SPOOLSTART system procedure's report-name parameter. CRE\_Spool\_Start\_ ensures that the string it passes to SPOOLSTART is 16 bytes long by truncating or adding blank characters to the string you specify.

*number\_of\_copies*

if present, specifies the number of copies to print. CRE\_Spool\_Start\_ passes *number\_of\_copies* to the num-of-copies SPOOLSTART parameter.

*page\_size*

if present, specifies the number of lines per page to be used by the Peruse utility for its PAGE and LIST commands. CRE\_Spool\_Start\_ passes *page\_size* to the page-size SPOOLSTART parameter.

*flags*

if present, specifies certain attributes for the spooler job. See the *Spooler Programmer's Guide* for details. CRE\_Spool\_Start\_ passes *flags* to the flags SPOOLSTART parameter.

*owner*

if present, specifies the spooler job owner. CRE\_Spool\_Start\_ passes *owner* to the owner SPOOLSTART parameter.

*max\_lines*

if present, specifies the maximum number of lines to allow for the spooler job. CRE\_Spool\_Start\_ passes *max\_lines* to the max-lines SPOOLSTART parameter.

*max\_pages*

if present, specifies the maximum number of pages to allow for the spooler job. CRE\_Spool\_Start\_ passes *max\_pages* to the max-pages SPOOLSTART parameter.

## Return Value

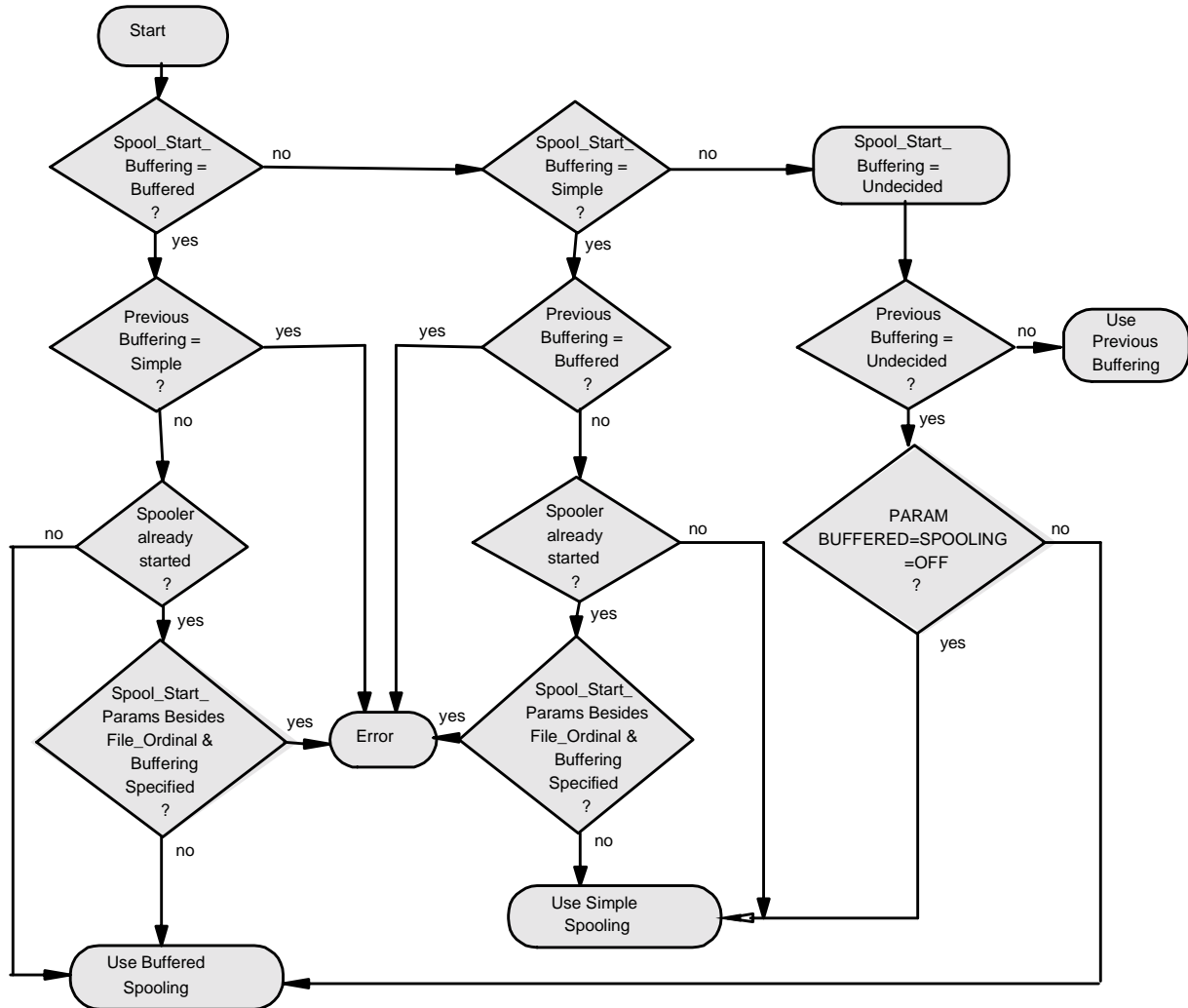
CRE\_Spool\_Start\_ returns one of the following:

- 0 if the input operation is successful
- A positive number, which is a file system error number
- A negative number, which is the negation of a CRE error number from the following table:

Error Code	Cause
55	Missing or invalid parameter
56	Invalid parameter value
57	Parameter value not accepted
63	Undefined shared file
64	File not open
65	Invalid attribute value
71	Inconsistent attribute value
108	Invalid I/O device

## Considerations

- CRE\_Spool\_Start\_ invokes the SPOOLSTART system procedure, passing the parameters it received to SPOOLSTART.
- CRE\_Spool\_Start\_ does not validate or alter any of the parameters you specify except to ensure that the lengths of the *location*, *form\_name*, and *report\_name* parameters are 16 bytes.
- The standard file specified by *file\_ordinal* must already be open.
- You can only reference the file associated with *file\_ordinal* with calls to CRE\_File\_Open\_ before you call CRE\_Spool\_Start\_.
- The file referenced by *file\_ordinal* must be a spooler collector. If your program calls CRE\_Spool\_Start\_ after it opens the file and before it calls any other procedure for the spooled file, the CRE uses the CRE\_Spool\_Start\_ *buffering* parameter to determine whether to use simple or buffered spooling.

**Figure 6-3. Determining Spooler Buffering in CRE\_Spool\_Start\_**

VST 603 .VSD

## \$RECEIVE Functions

This subsection describes the functions that support \$RECEIVE. Both the TNS and native CRE environment support \$RECEIVE functionality. [Table 6-4](#) on page 6-31 lists the functions in this subsection.

Table 6-4. \$RECEIVE Functions

Function Name	Function Action
<a href="#">CRE_Receive_Open_Close_</a> on page 6-31	Opens or closes \$RECEIVE.
<a href="#">CRE_Receive_Read_</a> on page 6-38	Reads a message from \$RECEIVE by calling READUPDATE.
<a href="#">CRE_Receive_Write_</a> on page 6-41	Replies to a message from \$RECEIVE by calling REPLYX.

CRE\_Receive\_Open\_Close\_

CRE\_Receive\_Open\_Close\_ provides a logical open or close of \$RECEIVE on behalf of its caller.

The syntax for the TNS environment is:

```
INT PROC CRE_Receive_Open_Close_(variant, attributes,
                                cplist) EXTENSIBLE;
    INT          variant;          ! in,      required
    INT          .EXT attributes;   ! in,      optional
    INT(32)      .cplist;           ! in/out, optional    TNS only
```

The syntax for the native environment is:

```
INT PROC CRE_Receive_Open_Close_(variant, attributes,
                                cplist) EXTENSIBLE;
    INT          variant;          ! in,      required
    INT          .EXT attributes;   ! in,      optional
    INT(32)      .EXT .cplist;      ! in/out, optional native only
```

*variant*

defines if the call is an open or a close request. It is nonzero for an open request, zero for a close request.

*attributes*

if present and its address value is not equal to zero, points to a structure that specifies attributes to apply to \$RECEIVE. For further information, see [Specifying the Receive File Open Attributes](#) on page 6-32.

*cplist*

if present and its address value is nonzero in the TNS environment, is a checkpoint list. In the native CRE environment, this parameter has no meaning and should be left empty (it is used by COBOL for process pair execution).

## Return Value

CRE\_Receive\_Open\_Close\_ returns one of the following:

- 0 if the input operation is successful
- A positive number, which is a file system error number
- A negative number, which is the negation of a CRE error number from the following table:

Error Code	Cause
17	Cannot obtain control space
55	Missing or invalid parameter
57	Parameter value not accepted
65	Invalid attribute value
71	Inconsistent attribute value

## Specifying the Receive File Open Attributes

This subsection describes the fields of the structure that the *attributes* parameter references.

The format of the *attributes* structure is defined by the following TAL STRUCT:

```
STRUCT CRE^RFOA^model( * );      ! $RECEIVE open attributes
BEGIN
  INT      Maximum^requesters;    ! Max requesters supported
  INT      Maximum^syncdepth;    ! Max saved / requester
  INT      Maximum^reply;        ! Max bytes in saved reply
  INT      Receive^depth;        ! Max server queue depth
  INT      Report^flags[0:3];    ! Report system msg flags
  FILLER 2;                      ! Reserved (must be 0)
END;  -- CRE^RFOA^model
```

The structure CRE^RFOA^model is in the CREDECS file. Refer to [Section 3, Compiling and Binding Programs for the TNS CRE](#), for information on the CREDECS file.

[Figure 6-4](#) on page 6-34 shows how each of the fields in CRE^RFOA^model affects how the CRE allocates memory for requester processes:

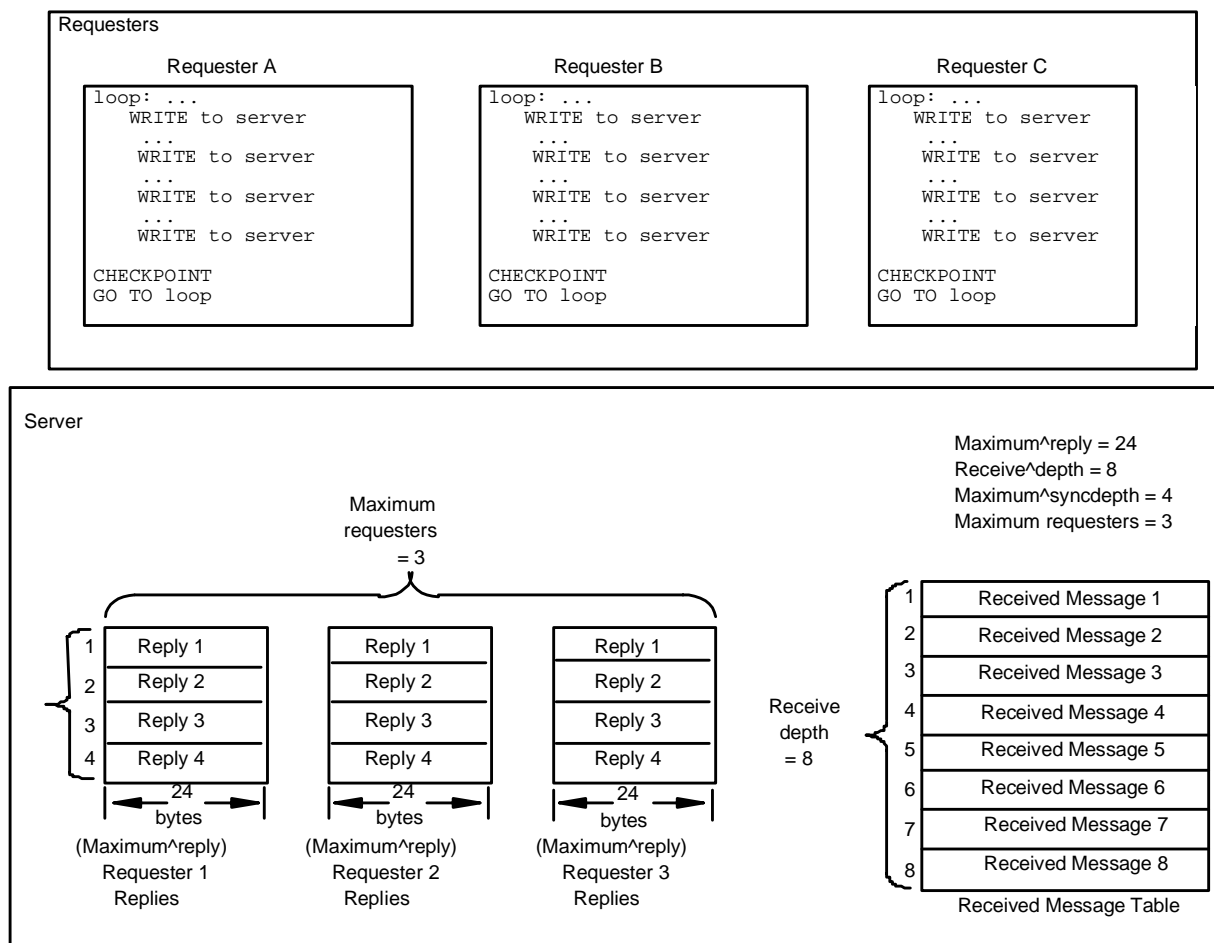
RFOA field	Effect
Maximum^requesters (value 3)	The server can support three simultaneous requester processes because it has allocated space for three sets of server replies: Requester 1 Replies, Requester 2 Replies, and Requester 3 Replies. Therefore, the server can support all three requesters: Requester A, Requester B, and Requester C.



<b>RFOA field</b>	<b>Effect (continued)</b>
Maximum^syncdepth (value 4)	The server can support requesters that issue up to four I/O requests to the server between calls to a checkpoint system procedure such as CHECKPOINT, CHECKPOINTX, CHECKPOINTMANYX, and so forth.
Maximum^reply (value 24 bytes)	The server can store replies that are up to 24 bytes long, as shown in each Reply table.
Receive^depth	The server's Received Message Table holds messages received from \$RECEIVE for which a REPLYX has not been issued. Because the table shows that it now holds eight messages, the operating system rejects any subsequent messages from requesters until the server calls the CRE_Receive_Write_ function to reply to one of the messages.

See the *Guardian Programmer's Guide* and the *Guardian Procedure Calls Reference Manual* for additional information about \$RECEIVE processing.

**Figure 6-4. Structure Allocation to Support Requesters Running as Process Pairs**



VST604.VSD

### Specifying Maximum^requesters:

- Maximum^requesters specifies the maximum number of opens from other processes that your process can manage at any point in time. The CRE returns error 12, "file in use," if it receives an open request that would exceed the number you specify in the Maximum^requesters field.

### Specifying Maximum^syncdepth:

- Maximum^syncdepth specifies the maximum number of replies that the CRE holds for each requester in case the requester's backup process becomes the primary process and begins executing the instructions that appear immediately after the previous CHECKPOINT. The value you specify for Maximum^syncdepth is application dependent. In general, Maximum^syncdepth specifies the maximum number of messages that any requester can send to your process before the requester calls a checkpoint system procedure such as CHECKPOINT,

CHECKPOINTX, CHECKPOINTMANY, CHECKPOINTMANYX, and so forth. Note that the CRE allocates enough space to accommodate the maximum number of retained replies times the maximum number of requesters times the maximum number of bytes per saved reply. For example:

```
Space_allocated := Maximum^requesters *
                  Maximum^syncdepth *
                  Maximum^reply;
```

Your program might be able to allocate less space if it can distinguish requests that alter a database from requests that only return information without altering a database. Your program might be able to allocate less space:

- If it does not retain messages that do not update a database.
- If it is not necessary that the program receive the same response to a database query for each such query. If it is possible for a database to be changed, however, by another process, two successive queries might return different and inconsistent responses.

Specifying `Maximum^reply`:

- `Maximum^reply` specifies the maximum number of bytes in each message that the CRE saves.

Specifying `Receive^depth`:

- You read messages by calling `CRE_Receive_Read_` and reply to messages by calling `CRE_Receive_Write_`. These functions correspond to the `READUPDTEX` and `REPLYX` system procedures.

Specifying `Report^flags`:

- You can specify, on a message by message basis, which system messages you want to receive. The CRE receives all system messages but returns to your program only those messages that you specify in `Report^flags`.

[Table 6-5](#) on page 6-36 shows literals that you use to specify the messages you want to receive. The literals shown in [Table 6-5](#) are in the CREDECS file.

**Table 6-5. Receive File Message Report Names**

<b>Symbolic Name</b>	<b>System Message Number</b>
Report^flags[0]	
CRE^cpu^down^mask	-2
CRE^cpu^up^mask	-3
CRE^settime^mask	-10
CRE^power^on^mask	-11
CRE^NEWPROCESSNOWAIT^mask	-12
CRE^message^missed^mask	-13
CRE^3270^status^mask	-21
CRE^SIGNALTIMEOUT^mask	-22
CRE^LOCKMEMORY^done^mask	-23
CRE^LOCKMEMORY^failed^mask	-24
CRE^PROCSIGNALTIMEOUT^mask	-26
Report^flags[0]	
CRE^CONTROL^mask	-32
CRE^SETMODE^mask	-33
CRE^RESETSYNC^mask	-34
CRE^CONTROLBUF^mask	-35
CRE^SETPARAM^mask	-37
CRE^message^cancelled^mask	-38
CRE^DEVICEINFO2^mask	-41
Report^flags[2]	
CRE^remote^cpu^down^mask	-100
CRE^process^deleted^mask	-101
CRE^PROCESS_CREATE_^mask	-102
CRE^OPEN^mask	-103
CRE^CLOSE^mask	-104
CRE^break^mask	-105
CRE^devinfo^query^mask	-106
CRE^subname^mask	-107
CRE^FILE_GETINFO_^mask	-108
CRE^FILENAME_FINDNEXT_^mask	-109
CRE^node^down^mask	-110
CRE^node^up^mask	-111
CRE^GMOM^notify^mask	-112
CRE^remote^cpu^up^mask	-113
Report^flags[3]	
CRE^pathsend^dialogue^abort^mask	-121

For each word in Report^flags, use a logical OR (LOR) operation to specify the system messages you want to receive. For example, your program is notified whenever the CRE receives a CONTROL, SETMODE, cpu^down, cpu^up, remote^cpu^down, remote^cpu^up, settime, power^on, OPEN, or CLOSE system message by initializing Report^flags, as follows:

```
STRUCT rcv_param (CRE^RFOA^model);

rcv_param.Report^flags[0] := 0;
rcv_param.Report^flags[1] := 0;
```

```

rcv_param.Report^flags[2] := 0;
rcv_param.Report^flags[3] := 0;

rcv_param.Report^flags[0] := CRE^cpu^down^mask      LOR
                           CRE^cpu^up^mask          LOR
                           CRE^settime^mask         LOR
                           CRE^power^on^mask        ;

rcv_param.Report^flags[1] := CRE^CONTROL^mask      LOR
                           CRE^SETMODE^mask        ;

rcv_param.Report^flags[2] := CRE^remote^cpu^down^mask LOR
                           CRE^remote^cpu^up^mask   LOR
                           CRE^OPEN^mask            LOR
                           CRE^CLOSE^mask           ;

```

If your program does not specify *attributes* or *attributes* is present but its address value is zero, the CRE uses the following *attributes*:

```

Maximum^requesters = 1
Maximum^syncdepth  = 1
Maximum^reply      = 0
Receive^depth      = 1
Report^flags[0:3]  = [-1, 0, 0, 0]

```

Report^flags[0] has special meaning. Refer to [Table 6-6](#) on page 6-37.

**Table 6-6. Using Report^flags**

Program Has Not Previously Opened \$RECEIVE		Program Has Already Opened \$RECEIVE	
Report^flags[0] < 0	Report^flags[0] => 0	Report^flags[0] < 0	Report^flags[0] => 0
Program does not receive system messages.	Program receives system messages specified in Report^flags[0:3].	Caller receives same system messages as original opener of \$RECEIVE.	If caller specifies the same system messages as original opener, Report^flags are okay. Otherwise, CRE_Receive_Open_Close_ returns an error.

## Specifying the Close Variant

If *variant* specifies 0 to close a connection, CRE\_Receive\_File\_Open\_Close\_ closes the specified connection. If the number of connections becomes zero but the CRE has messages queued that have not been read by the program, CRE\_Receive\_Open\_Close\_ returns a 201 reply to each of the messages. The CRE\_Receive\_Open\_Close\_ closes the file unless it needs to maintain an open for internal reasons such as monitoring a backup process.

## CRE\_Receive\_Read\_

CRE\_Receive\_Read\_ calls the READUPDATE system procedure to request a record from \$RECEIVE.

```

INT PROC CRE_Receive_Read_( buffer:read_count, count_read,
                           time_out, sender_info, flags )
                           EXTENSIBLE;
  STRING  .EXT buffer;           ! out,   required
  INT      read_count;           ! in,   required
  INT      .EXT count_read;       ! out,   required
  INT(32)   time_out;            ! in,   optional
  INT      .EXT sender_info;      ! out,   optional
  INT      flags;                ! in, optional TNS, native

```

*buffer:read\_count*

is the data area (*buffer*) in which to store the record read and the maximum number of bytes to read (*read\_count*).

*count\_read*

is assigned the number of bytes read if a record is obtained.

*time\_out*

if present, specifies how many hundredths of a second CRE\_Receive\_Read\_ should wait for a message from \$RECEIVE before timing out the request.

If *time\_out* is greater than or equal to zero, the CRE returns error 40, time out, if it does not receive a message from \$RECEIVE within *time\_out* hundredths of a second.

If *time\_out* is less than zero or you do not specify *time\_out*, CRE\_Receive\_Read\_ does not time out and waits as long as necessary to receive a message on \$RECEIVE.

*sender\_info*

if present and its value is not equal to zero, is a pointer to a structure, called a CRE^sender^model. CRE\_Receive\_Read\_ stores in CRE^sender^model a description of the process that sent the current message.

*sender\_info* must be a pointer to a structure with the following layout:

```

STRUCT CRE^sender^model( * );
BEGIN
  INT      System^flag;           ! -1 => system, 0 => user
  INT      Entry^number;          ! Sender's requester number
  INT      Message^number;        ! Used for message queuing
  INT      File^number;           ! Sender's open number
  INT      Phandle[0:9];          ! Sender's phandle
  INT      Read^count;            ! Read count from WRITEREAD

```

```
INT      Dialog^Flags;      ! See following description
END;    ! CRE^sender^model
```

`Dialog^Flags` is used by context-sensitive Pathway servers. It provides dialog information about the server-class send operation that a requestor initiated by means of a `Pathsend` procedure call.

The bits of `Dialog^Flags` have the following meanings:

- <0:11>    Reserved
- <12:13>   Dialog status. Indicates the last operation performed by the message sender. Values are:
  - 0 Context-free server-class send operation.
  - 1 First server-class end operation in a new dialog.
  - 2 Server-class send operation in an existing dialog.
  - 3 Aborted dialog. No further server-class send operations will be received in this dialog. There is no buffer associated with this value.
- <14>      This is a copy of the *flags*.<14> parameter bit in the requester's call to the `Pathsend SERVERCLASS_DIALOG_BEGIN_` procedure. This bit identifies the transaction model the requester is using for dialogs.
- <15>      Reserved

For more information about the use of `Pathsend` procedures, `Pathsend` dialogs, and the `Dialog^Flags` field, see the *NonStop TS/MP Pathsend and Server Programming Manual*.

*flags*

if present, is the “any requesters” flag setting if an end-of-file is encountered (no messages are waiting in `$RECEIVE` or the requester is waiting for an open request). If *flags* is not present, its value is treated as zero.

## Using CRE\_Receive\_Read\_

`CRE_Receive_Read_` reads messages from `$RECEIVE` by calling the `READUPDATE` system procedure. If your program specifies `Receive^depth` equals one when it calls `CRE_Receive_Open_Close_` and a received message requires a reply, `CRE_Receive_Read_` calls `CRE_Receive_Write_` to reply automatically to the message. If your program specifies `Receive^depth` greater than one, but the CRE's table of unreplied messages is full, `CRE_Receive_Read_` returns error 74 (the number of `READUPDATE`s without replies exceeds the `Receive^depth` specified) to the process that called your process.

## Handling Generic System Messages

Generic system messages describe changes in the environment in which your program is running such as a SETTIME message or a CPU down message. Generic system messages do not apply to a specific requester although they might affect your process's requesters. The CRE updates its state information if it receives a generic system message. For example, the CRE updates its requester information if it receives a CPU down message for a CPU in which one of your process's requesters is running. In addition, the CRE passes the system message to your program if you included the mask for that message in Report^flags when you called CRE\_Receive\_Open\_Close\_. See "CRE\_Receive\_Open\_Close\_" in this section for more details.

## Handling System Messages and Requester Messages

Your process receives:

- System messages that pertain directly to requesters that have opened your process.

For example, when a requester calls the CONTROL, CONTROLBUF, RESETSYNC, SETMODE, or SETPARAM system procedure, the operating system sends a system message to your program with the information specified by the requester. If the requester is not known to the CRE, CRE\_Receive\_Read\_ rejects the request without passing information to your program.

If the CRE receives a system message from a known requester, it returns the message to your program if you specified the type of message received in Report^flags when you called CRE\_Receive\_Open\_Close\_.

The CRE always returns nonsystem messages from known requesters to your program.

- Open messages from requesters attempting to open your program as a server.

If the CRE receives an "open" message from a requester (a message that another process is attempting to open your program) and your program specifies the CRE^OPEN^mask as a Report^flag when it calls CRE\_Receive\_Open\_Close\_, CRE\_Receive\_Read\_ passes the open message to your program. Your program must call CRE\_Receive\_Write\_ to specify whether to honor the open request or refuse it.

---

**Note.** If another process issues a nowait open to your process and you have specified that you want to receive "open" messages, the other process cannot continue opening your program until you have responded to the open message. If the other process is opening your program with a waited open, the other process remains suspended entirely until your program replies to the open request. Therefore, you might want to ensure that your program responds quickly to an open request.

---

See the *reply\_code* parameter to the CRE\_Receive\_Write\_ function for more information.



## Return Value

CRE\_Receive\_Read\_ returns one of the following:

- 0 if the input operation is successful
- A positive number, which is a file system error number
- A negative number, which is the negation of a CRE error number from the following table:

Error Code	Cause
55	Missing or invalid parameter
64	File not open

## CRE\_Receive\_Write\_

CRE\_Receive\_Write\_ calls the REPLYX system procedure to reply to a message from \$RECEIVE.

```

INT PROC CRE_Receive_Write_( buffer:write_count, reply_code,
                             message_number )
                             EXTENSIBLE;
    STRING .EXT buffer;           ! in, required
    INT        write_count;       ! in, required
    INT        reply_code;        ! in, required
    INT        message_number;    ! in, required  TNS, native

```

*buffer:write\_count*

is the data area (*buffer*) from which to write and the number of bytes to write (*write\_count*).

*reply\_code*

is the reply code that REPLYX returns to the requester process. The value you specify for *reply\_code* determines the value of the condition code indicator (CCL, CCE, or CCG), and is the value returned if the requester process calls FILEINFO or one of the FILEGETINFO system procedures. See the *Guardian Procedure Calls Reference Manual* for more information on FILEINFO and the FILEGETINFO procedures.

*message\_number*

specifies the CRE message number to which the reply corresponds. If the value you specify for Reply^depth in the CRE\_Receive\_Open\_Close\_ *attributes* parameter is greater than one, you must specify *message\_number* when you call CRE\_Receive\_Write\_. CRE\_Receive\_Read\_ returns the message number to your program in the *message\_number* field of its *sender\_info* parameter.

If you do not specify `reply^depth` when you call `CRE_Receive_Open_Close_` or you specify a depth of one, you must specify zero for the `message_number` parameter.

Return Value

`CRE_Receive_Write_` returns one of the following:

- 0 if the input operation is successful
- A positive number, which is a file system error number
- A negative number, which is the negation of a CRE error number from the following table:

Error Code	Cause
55	Missing or invalid parameter
64	File not open

CRE\_Terminator\_

The `CRE_Terminator_` procedure ensures that all standard files in your program are closed. In the TNS CRE environment, it also calls a language-specific function from the run-time library of each language represented by a module in your program. Language-specific functions are not called in the native CRE environment.

```
PROC CRE_Terminator_( completion_status, options,
                      completion_code, termination_info,
                      spi_ssid, text:text_length )
    EXTENSIBLE;
    INT      completion_status;    ! in, required
    INT      options;              ! in, optional
    INT      completion_code;      ! in, optional
    INT      termination_info;     ! in, optional
    INT      .EXT spi_ssid;         ! in, optional
    STRING   .EXT text;            ! in, optional
    INT      text_length;          ! in, optional    TNS, native
```

*completion\_status*

describes your program's status when it ended execution. Valid values for *completion\_status* are:

- CRE^Completion^normal
- CRE^Completion^warning
- CRE^Completion^error
- CRE^Completion^trap
- CRE^Completion^fatal

The meanings of completion codes are:

`CRE^Completion^normal`

specifies that your program completed either by reaching the end of its main routine or by invoking a language-specific normal termination verb such as `STOP RUN` in COBOL.

`CRE^Completion^warning`

specifies that your program reported warnings during its execution.

`CRE^Completion^error`

specifies that termination was due to a logic error in your program.

`CRE^Completion^trap`

specifies that termination was due to a trap (other than an arithmetic overflow trap, for which `CRE^Completion^error` is the correct completion status).

`CRE^Completion^fatal`

specifies that the CRE or a run-time library detected an internal error (for example, data corruption) within its own environment.

*options*

if present, is passed to the `PROCESS_STOP_` system procedure as its `options` parameter. If you do not specify *options*, the CRE derives a value. See [Table 6-7](#) on page 6-44 for derived values. Refer to the `PROCESS_STOP_` procedure in the *Guardian Procedure Calls Reference Manual* for more information about the *options* parameter.

*completion\_code*

if present, is passed to the `PROCESS_STOP_` system procedure as its completion-code parameter. If you do not specify *completion\_code*, the CRE derives a value. See [Table 6-7](#) on page 6-44 for derived values. Refer to the `PROCESS_STOP_` procedure in the *Guardian Procedure Calls Reference Manual* for more information about the *completion\_code* parameter.

*termination\_info*

if present, is passed to the `PROCESS_STOP_` system procedure for its termination-info parameter. Refer to the `PROCESS_STOP_` procedure in the *Guardian Procedure Calls Reference Manual* for more information about the *termination\_info* parameter.

*spi\_ssaid*

is an optional reference parameter. If present, its address is passed to the PROCESS\_STOP\_ system procedure for its spi-ssid parameter. *spi\_ssaid* must reference a valid 12-character subsystem identifier (ssid). See PROCESS\_STOP\_ in the *Guardian Procedure Calls Reference Manual* for more information. Refer to the PROCESS\_STOP\_ procedure in the *Guardian Procedure Calls Reference Manual* for more information about the *spi\_ssaid* parameter.

*text:text\_length*

is an optional parameter pair. If present, the string referenced by *text:text\_length* is passed to the PROCESS\_STOP\_ system procedure for its text:length parameter pair. *text* and *text\_length* must both be present or omitted. Refer to the PROCESS\_STOP\_ procedure in the *Guardian Procedure Calls Reference Manual* for more information about the *text:text\_length* parameter.

In the TNS CRE environment, CRE\_Terminator\_ closes your program's standard files, calls a language-specific termination function for each language represented in your program, and calls the PROCESS\_STOP\_ system procedure.

In the native CRE environment, CRE\_Terminator\_ closes your program's standard files. It then calls a NonStop operating system routine, which calls all standard run-time library termination routines and global destructors.

For more information about program termination, refer to [Section 2, CRE Services](#).

**Table 6-7. Default Values for Options and Completion Code**

completion_ status	Normal Termination		Abnormal Termination	
	Options Specified and Options.<15> = 0	completion_ code	Options Not Specified or Options.<15> = 1	completion_ code
normal	0	none		none
warning	0	1		1
error	1	none		3
trap	1	none		3
fatal	1	none		5

# Exception-Handling Functions

This subsection describes the functions that support exception handling. [Table 6-8](#) on page 6-45 lists the functions in this subsection.

**Table 6-8. Exception-Handling Functions**

Function Name	Function Action
<a href="#">CRE_Log_GetPrefix_</a> on page 6-45	Returns to its caller the text that precedes each message in standard log.
<a href="#">CRE_Stacktrace_</a> on page 6-45	Writes a stack trace to either a standard file or to an operating system file.

## CRE\_Log\_GetPrefix\_

CRE\_Log\_GetPrefix\_ returns to its caller the default prefix text that precedes each line of error or warning information written to standard log.

```
INT PROC CRE_Log_GetPrefix_(buffer:buffer_bytes,
                           prefix_bytes) EXTENSIBLE;
  STRING .EXT buffer;           ! in/out, optional
  INT     buffer_bytes;         ! in, optional
  INT     .EXT prefix_bytes;    ! out, optional TNS,native
```

*buffer* : *buffer\_bytes*

if present, specifies the address and size of the buffer that CRE\_Log\_GetPrefix\_ fills with the default prefix text. CRE\_Log\_GetPrefix\_ stores the prefix text only if both *buffer* and *buffer\_bytes* are present.

*prefix\_bytes*

if present, is the actual number of bytes in the prefix text.

## Return Value

CRE\_Log\_GetPrefix\_ returns 0 if *buffer:buffer\_bytes* contains prefix text. Otherwise, CRE\_Log\_GetPrefix\_ returns the value returned to it by the PROCESS\_GETINFO\_ system procedure.

## Considerations

If either *buffer* or *buffer\_bytes* is not passed, CRE\_Log\_GetPrefix\_ returns 0 in *prefix\_bytes* and leaves *buffer* unchanged.

## CRE\_Stacktrace\_

CRE\_Stacktrace\_ writes a stack trace to a standard file.

The syntax for the TNS environment is:

```
INT PROC CRE_Stacktrace_( file_ordinal, buffer:buffer_bytes,
                          prefix_bytes, stack_mark )
                          EXTENSIBLE;
  INT      file_ordinal;    ! in,      optional
  STRING .EXT buffer;       ! in/out,  optional
  INT      buffer_bytes;    ! in,      optional
  INT      prefix_bytes;    ! in,      optional
  INT      .stack_mark;     ! in,      optional   TNS only
```

The syntax for the native CRE environment is:

```
INT PROC CRE_Stacktrace_( file_ordinal, buffer:buffer_bytes,
                          prefix_bytes ) EXTENSIBLE;
  INT      file_ordinal;    ! in,      optional
  STRING .EXT buffer;       ! in/out,  optional
  INT      buffer_bytes;    ! in,      optional
  INT      prefix_bytes;    ! in,      optional   native only
```

*file\_ordinal*

if present, identifies an open standard file. If absent, CRE\_Stacktrace\_ writes trace information to standard log. You can use the following symbolic names for *file\_ordinal*:

CRE File Ordinal Symbolic Names	
CRE^Standard^Output	
CRE^Standard^Log	

*buffer:buffer\_bytes*

specifies the address and size of the buffer in which to compose stack trace report lines. The buffer must be a minimum of 132 characters. CRE\_Stacktrace\_ uses a 132-byte local buffer if either *buffer* or *buffer\_bytes* is not present.

*prefix\_bytes*

if present, requests that each line include a prefix before the generated text. If absent, or if CRE\_Stacktrace\_ uses a local buffer, CRE\_Stacktrace\_ uses zero for *prefix\_bytes*. *prefix\_bytes* specifies how many bytes of prefix text to use with each lines of the stack trace:

<i>prefix_bytes</i>	Meaning
< -1	All lines of stack trace include the first <i>prefix_bytes</i> of <i>buffer:buffer_bytes</i> as prefix text.

<i>prefix_bytes</i>	Meaning (continued)
-1	Do not include prefix bytes with any line of stack trace.
0	Do not include prefix bytes with any line of stack trace.
> 0	Include the first <i>prefix_bytes</i> of <i>buffer:buffer_bytes</i> on the first line of stack trace and <i>prefix_bytes</i> blanks on each subsequent line of stack trace.

*stack\_mark*

(TNS CRE only) if present, specifies the L-register word of the first stack marker to display in the stack trace. If *stack\_mark* is not present, reporting begins with the caller of the caller of CRE\_Stacktrace\_.

## Return Value

CRE\_Stacktrace\_ returns one of the following:

- 0 if the input operation is successful
- A positive number, which is a file system error number
- A negative number, which is the negation of a CRE error number from the following table:

Error Code	Cause
56	Invalid parameter value
63	Undefined shared file
64	File not open

## Considerations

- If a line of stack trace is longer than the record size for the output device, CRE\_Stacktrace\_ breaks the message over as many lines as necessary.
- CRE\_Stacktrace\_ retries errors that are recoverable.
- CRE\_Stacktrace\_ begins examining the names of procedures on the stack beginning with the current procedure—the most-recently called procedure. From the current procedure, it begins scanning stack frames until it encounters a procedure name that does not end in an underscore character (\_). When CRE\_Stacktrace\_ encounters a procedure name that does not end in an underscore, it writes the stack information for that procedure to the file specified by *file\_ordinal*. Thereafter, CRE\_Stacktrace\_ includes all procedures in the stack trace, including procedures whose names end in an underscore.





# 7 Math Functions

This section describes the interfaces to the math functions. It describes:

- [Arithmetic Overflow Handling](#) on page 7-1
- [Standard Math Functions](#) on page 7-1
- [Sixty-Four-Bit Logical Operations \(Bit Manipulation Functions\)](#) on page 7-23
- [Remainder](#) on page 7-25
- [Decimal Conversion Functions](#) on page 7-25

Each function described in this section begins with the prefix RTL\_ or CRE\_. Refer to [Using Standard Functions](#) on page 2-56 for more information.

The pTAL RTL functions, data, and data structure declarations are in the RTLRDECS file.

## Arithmetic Overflow Handling

In the TNS CRE environment, functions that can cause an arithmetic overflow specify that a fault might occur, rather than specifying that a trap might occur. A TNS CRE or RTL function that can cause an arithmetic overflow disables overflow traps while it executes and tests explicitly for arithmetic overflow after each instruction that can cause it. Prior to returning, the function enables the arithmetic overflow bit in the environment register image of the most recent stack marker. If traps were enabled when your program called the function, your program will trap immediately upon return from the function. If traps were disabled when your program called the function, your program will not trap. You might be able to test explicitly for arithmetic overflow upon return from the function. Refer to the language manual for your routine that calls the function for more information on detecting arithmetic overflow.

The native CRE environment provides only non-trapping variants for functions that can cause an arithmetic trap. Non-trapping variants dynamically detect an arithmetic fault, set `errno`, and return an appropriate value.

Since the native CRE architecture does not have a saved environment register, you must evaluate `errno` upon return from the function. Refer to the language manual for your routine that calls the function for more information on detecting arithmetic overflow.

## Standard Math Functions

This subsection describes the standard math functions. [Table 7-1](#) on page 7-2 summarizes the standard math functions.

**Table 7-1. TNS CRE Standard Math Functions** (page 1 of 2)

<b>Function Name</b>	<b>Function Action</b>
<a href="#">Arccos</a> on page 7-4	Returns an arccosine expressed in radians.
<a href="#">Arcsin</a> on page 7-5	Returns an arcsine expressed in radians.
<a href="#">Arctan</a> on page 7-5	Returns an arctangent expressed in radians.
<a href="#">Arctan2</a> on page 7-6	Returns the arctangent, expressed in radians, of the quotient of its parameters.
<a href="#">Cos</a> on page 7-7	Returns the cosine of an angle whose size is expressed in radians.
<a href="#">Cosh</a> on page 7-7	Returns a hyperbolic cosine.
<a href="#">Exp</a> on page 7-8	Returns the exponential function (base e) of its parameter.
<a href="#">Ln</a> on page 7-8	Returns a natural (base e) logarithm.
<a href="#">Log10</a> on page 7-9	Returns a common (base 10) logarithm.
<a href="#">Lower</a> on page 7-10	Returns the largest integer that is not greater than the value of its parameter.
<a href="#">Mod</a> on page 7-11	Returns the remainder of an integer division calculation.
<a href="#">Normalize</a> on page 7-12	Splits a floating-point number into a normalized fraction and an integral power of two.
<a href="#">Odd</a> on page 7-13	Determines whether a value is even or odd.
<a href="#">Positive_Diff</a> on page 7-13	Returns the arithmetic difference between two numbers.
<a href="#">Power</a> on page 7-15	Returns a number raised to a specified power.
<a href="#">Power2</a> on page 7-16	Multiplies a number by an integral power of two.
Random_set and Random_next on page <a href="#">7-17</a>	Establishes the seed—the initial value—in a pseudo-random number sequence.
<a href="#">Round</a> on page 7-17	Returns a pseudo-random number.
	Returns the nearest whole number.

**Table 7-1. TNS CRE Standard Math Functions** (page 2 of 2)

Function Name	Function Action
<a href="#">Sign</a> on page 7-18	Returns its first parameter with the sign set according to its second parameter.
<a href="#">Sin</a> on page 7-19	Returns the sine of an angle whose size is expressed in radians.
<a href="#">Sinh</a> on page 7-19	Returns a hyperbolic sine.
<a href="#">Split</a> on page 7-20	Separates a floating-point number into integral and fractional parts.
<a href="#">Sqrt</a> on page 7-20	Returns the square root of a number.
<a href="#">Tan</a> on page 7-21	Returns the tangent of an angle whose size is expressed in radians.
<a href="#">Tanh</a> on page 7-21	Returns a hyperbolic tangent.
<a href="#">Truncate</a> on page 7-22	Returns the nonfractional part of a number.
<a href="#">Upper</a> on page 7-22	Returns the smallest integer that is not less than the value of its parameter.

[Table 7-2](#) on page 7-3 shows the standard math functions supported by the native CRE library. The native CRE does not provide pTAL prototypes for these functions. It provides only the function names, which are case-sensitive. You can use the function prototypes in the C header files as examples when writing your own pTAL prototypes for these functions.

See the *Guardian Native C Library Calls Reference Manual* or the *Open System Services Library Calls Reference Manual* for descriptions of these functions.

**Table 7-2. Native CRE Standard Math Functions** (page 1 of 2)

Function Name	Native CRE Library Function Name
acos	Returns an arccosine expressed in radians
acosh	Returns a hyperbolic arccosine
asin	Returns an arcsine expressed in radians
asinh	Returns a hyperbolic arcsine
atan	Returns an arctangent expressed in radians
atan2	Returns the arctangent, expressed in radians, of the quotient of its parameters
cbt	Returns a cube root

**Table 7-2. Native CRE Standard Math Functions** (page 2 of 2)

Function Name	Native CRE Library Function Name
ceil	Returns the smallest integer that is not less than the value of its parameter
cos	Returns the cosine of an angle whose size is expressed in radians
cosh	Returns a hyperbolic cosine
erf	Returns the error function of x
erfc	Returns the complementary error function of x
exp	Returns the exponential function (base e) of its parameter
floor	Returns the largest integer that is not greater than the value of its parameter
fmod	Returns the modulo remainder
frexp	Breaks a floating-point number into a fraction and a power of 2
log	Returns a natural (base e) logarithm
log10	Returns a common (base 10) logarithm
mod	Returns the remainder of an integer division calculation
modf	Separates a floating-point number into integral and fractional parts
sin	Returns the sine of an angle whose size is expressed in radians
sinh	Returns a hyperbolic sine
sqrt	Returns the square root of a number
tan	Returns the tangent of an angle whose size is expressed in radians
tanh	Returns a hyperbolic tangent

## Arccos

The Arccos functions return the arccosine of their parameter.

```

REAL(32) PROC { CRE_Arccos_Real32_ } ( cos );
               { RTL_Arccos_Real32_ }
REAL(32) cosi ! in TNS only

REAL(64) PROC { CRE_Arccos_Real64_ } ( cos );
               { RTL_Arccos_Real64_ }
REAL(64) cosi ! in TNS only

```

*cos*

is a number in the range:

-1.0 less than or equal to *cos* less than or equal to 1.0

An arithmetic fault occurs if *cos* is not within this range.

## Return Value

The Arccos functions return the angle, expressed in radians, whose cosine is *cos*. The value returned is in the range

0 less than or equal to  $\arccos(\textit{cos})$  less than or equal to  $\pi$

## Arcsin

The Arcsin functions return the arcsine of their parameter.

REAL(32) PROC { CRE_Arcsin_Real32_ }	( <i>sin</i> );	
REAL(32) <i>sin</i> ;	! in	TNS only
REAL(64) PROC { CRE_Arcsin_Real64_ }	( <i>sin</i> );	
REAL(64) <i>sin</i> ;	! in	TNS only

*sin*

is a number in the range:

-1.0 less than or equal to *sin* less than or equal to 1.0

An arithmetic fault occurs if *sin* is not within this range.

## Return Value

The Arcsin functions return the angle, expressed in radians, whose sine is *sin*. The value returned is in the range:

$-\pi/2$  less than or equal to  $\arcsin(\textit{sin})$  less than or equal to  $\pi/2$

## Arctan

The Arctan functions return the arctangent of their parameter.

REAL(32) PROC RTL_Arctan_Real32_( <i>tan</i> );	
REAL(32) <i>tan</i> ;	! in TNS only
REAL(64) PROC RTL_Arctan_Real64_( <i>tan</i> );	
REAL(64) <i>tan</i> ;	! in TNS only

*tan*

is the number whose arctangent is returned.

## Return Value

The Arctan functions return the angle, expressed in radians, whose tangent is  $\tan$ . The value returned is in the range:

$-\pi/2$  less than or equal to  $\arctan(\tan)$  less than or equal to  $\pi/2$

## Arctan2

The Arctan2 functions return the arctangent of the quotient of their parameters.

<pre> REAL(32) PROC { CRE_Arctan2_Real32_ } ( y, x );                 { RTL_Arctan2_Real32_ } REAL(32) y, x;                                ! in                TNS only </pre>
<pre> REAL(64) PROC { CRE_Arctan2_Real64_ } ( y, x );                 { RTL_Arctan2_Real64_ } REAL(64) y, x;                                ! in                TNS only </pre>

$y$

is the ordinate of the ordered pair  $(y, x)$  that determines an angle whose tangent is  $y/x$ .

$x$

is the abscissa of the ordered pair  $(y, x)$  that determines an angle whose tangent is  $y/x$ .

## Return Value

The Arctan2 functions return the angle, expressed in radians, whose tangent is  $y/x$ . The value returned is in the range:

$-\pi < \arctan2(y, x)$  less than or equal to  $\pi$

The quadrant of the angle,  $A$ , returned by the Arctan2 functions is determined by the signs of  $y$  and  $x$ :

if  $y < 0$  then  $A < 0$

if  $x < 0$  then  $\text{absolute\_value}(A) > \pi/2$

if  $x = 0$  then  $\text{absolute\_value}(A) = \pi/2$

## Considerations

An arithmetic fault occurs if both  $y$  and  $x$  are equal to zero.

## Cos

The Cos functions return the cosine of an angle.

REAL(32) PROC RTL_Cos_Real32_( <i>angle</i> ); REAL(32) <i>angle</i> ;	! in	TNS only
REAL(64) PROC RTL_Cos_Real64_( <i>angle</i> ); REAL(64) <i>angle</i> ;	! in	TNS only

*angle*

is the angle, expressed in radians, whose cosine is returned.

### Return Value

The Cos functions return the cosine of *angle*. The cosine is in the range:

-1.0 less than or equal to cos( *angle* ) less than or equal to 1.0

## Cosh

The Cosh functions return a hyperbolic cosine.

REAL(32) PROC { CRE_Cosh_Real32_ RTL_Cosh_Real32_ } ( <i>number</i> ); REAL(32) <i>number</i> ;	! in	TNS only
REAL(64) PROC { CRE_Cosh_Real64_ RTL_Cosh_Real64_ } ( <i>number</i> ); REAL(64) <i>number</i> ;	! in	TNS only

*number*

is the number whose hyperbolic cosine is returned.

### Return Value

The Cosh functions return the hyperbolic cosine of *number*. The value returned is:

$$\frac{\exp(\textit{number}) + \exp(-\textit{number})}{2}$$

### Considerations

An arithmetic fault occurs if an intermediate calculation causes an overflow.

## Exp

The Exp functions return e—the natural logarithm base—raised to the power passed as a parameter to Exp.

REAL(32) PROC { CRE_Exp_Real32_ }	( <i>number</i> );	
{ RTL_Exp_Real32_ }		
REAL(32) <i>number</i> ;	! in	TNS only
REAL(64) PROC { CRE_Exp_Real64_ }	( <i>number</i> );	
{ RTL_Exp_Real64_ }		
REAL(64) <i>number</i> ;	! in	TNS only

*number*

is the exponent to which e is raised.

## Return Value

The Exp functions return

$e^{\text{number}}$

where e is the natural logarithm base, approximately 2.718281828459045.

## Considerations

An arithmetic fault occurs if an intermediate calculation causes an overflow or underflow.

## Ln

The Ln functions return a natural—that is, base e—logarithm.

REAL(32) PROC { CRE_Ln_Real32_ }	( <i>number</i> );	
{ RTL_Ln_Real32_ }		
REAL(32) <i>number</i> ;	! in	TNS only
REAL(64) PROC { CRE_Ln_Real64_ }	( <i>number</i> );	
{ RTL_Ln_Real64_ }		
REAL(64) <i>number</i> ;	! in	TNS only

*number*

is the number whose natural logarithm is returned.

## Return Value

The Ln functions return “exponent,” where exponent satisfies the equation:

$$e^{\text{exponent}} = \text{number}$$



$e$  is the natural logarithm base, approximately 2.718281828459045.

## Considerations

An arithmetic fault occurs if *number* is less than or equal to 0.

## Examples

```
REAL(32) r1, r2, r3;
r1 := CRE_Ln_Real32_( 7.389056096 ); ! r1 is approx 2.
r2 := CRE_Ln_Real32_( 20.08553691 ); ! r2 is approx 3.
r3 := CRE_Ln_Real32_( 54.59814999 ); ! r3 is approx 4.
```

The values returned in R1, R2, and R3 derive from the following:

$\ln(7.389056096) \cong 2$ . That is,  $e^2 \cong 7.389056096$

$\ln(20.08553691) \cong 3$ . That is,  $e^3 \cong 20.08553691$

$\ln(54.59814999) \cong 4$ . That is,  $e^4 \cong 54.59814999$

## Log10

The Log10 functions return a common—that is, base 10—logarithm.

REAL(32) PROC { CRE_Log10_Real32_ { RTL_Log10_Real32_ } ( number ); REAL(32) number; ! in	TNS only
REAL(64) PROC { CRE_Log10_Real64_ { RTL_Log10_Real64_ } ( number ); REAL(64) number; ! in	TNS only

*number*

is the number whose common logarithm is returned.

## Return Value

The Log10 functions return “exponent,” where exponent satisfies the equation:

$$10^{\text{exponent}} = \text{number}$$

## Considerations

An arithmetic fault occurs if *number* is less than or equal to 0.

## Examples

```
REAL(32) r1, r2, r3;
r1 := CRE_Log10_Real32_(100); ! r1 is approx 2.
r2 := CRE_Log10_Real32_(1000); ! r2 is approx 3.
r3 := CRE_Log10_Real32_(10000); ! r3 is approx 4.
```

The values returned in R1, R2, and R3 derive from the following:

$\log_{10}(100) = 2$ . That is,  $10^2 = 100$

$\log_{10}(1000) = 3$ . That is,  $10^3 = 1000$

$\log_{10}(10000) = 4$ . That is,  $10^4 = 10000$

## Lower

The Lower function returns the largest integer that is not greater than the value of its parameter.

<pre>REAL(64) PROC RTL_Lower_Real64_( <i>number</i> );     REAL(64) <i>number</i>;                ! in</pre>	TNS only
--	----------

*number*

is the number for which Lower returns its value.

## Return Value

Lower returns the largest integer that is not greater than *number*.

## Example

```
REAL(64) r;
```

```
r := RTL_Lower_Real64_(1.8L0);      ! r gets 1.0L0
r := RTL_Lower_Real64_(-1.8L0);    ! r gets -2.0L0
```

## Mod

The Mod functions return the result of dividing their first parameter by their second parameter using integer division if their parameters are integers, or using floating-point division if their parameters are floating-point numbers.

INT	PROC { CRE_Mod_Int16_ RTL_Mod_Int16_ }	( <i>number</i> , <i>modulus</i> );	
INT	<i>number</i> , <i>modulus</i> ;	! in	TNS only
INT(32)	PROC { CRE_Mod_Int32_ RTL_Mod_Int32_ }	( <i>number</i> , <i>modulus</i> );	
INT(32)	<i>number</i> , <i>modulus</i> ;	! in	TNS only
INT(64)	PROC { CRE_Mod_Int64_ RTL_Mod_Int64_ }	( <i>number</i> , <i>modulus</i> );	
INT(64)	<i>number</i> , <i>modulus</i> ;	! in	TNS only
REAL(32)	PROC RTL_Mod_Real32_ REAL(32)	( <i>number</i> , <i>modulus</i> );	
	<i>number</i> , <i>modulus</i> ;	! in	TNS only
REAL(64)	PROC RTL_Mod_Real64_ REAL(64)	( <i>number</i> , <i>modulus</i> );	
	<i>number</i> , <i>modulus</i> ;	! in	TNS only

*number*

is the dividend in calculating the remainder.

*modulus*

is the divisor in calculating the remainder.

## Return Value

For an integer modulo function, the Mod functions return the result of evaluating the following expression using integer division:

$$\text{number} - \left( \frac{(\text{number})}{(\text{modulus})} * \text{modulus} \right)$$

For a floating-point modulo function, the Mod functions return the result of evaluating the following expression using floating-point division:

$$\text{number} - \left( \frac{\text{truncate}(\text{number})}{(\text{modulus})} * \text{modulus} \right)$$

## Considerations

If *modulus* = 0:

- The integer modulo functions causes an arithmetic fault.
- The floating-point modulo functions return 0.

## Example

```

INT i, j, k;
REAL(32) r, s, t;

i := 17;
j := 5;
k := RTL_Mod_Int16_(i, j); ! k gets 2

r := 17.2E0;
s := 0.5E0;
t := RTL_Mod_Real32_(r, s); ! t gets 0.2E0

```

## Normalize

The Normalize function splits a floating-point number into a normalized fraction and an integer power of two. This function is not available in the native CRE library.

REAL(64) PROC RTL_Normalize_Real64_( <i>number</i> , <i>power</i> );		
REAL(64) <i>number</i> ;	! in	
INT .EXT <i>power</i> ;	! out	TNS only

*number*

is the number to split.

*power*

is a reference parameter into which Normalize stores the integer power of 2.

## Return Value

Normalize returns a number, *y*, such that:

0.5 is less than or equal to  $y < 1$  and  $number = y * 2^{power}$

If *number* = 0, Normalize returns 0 for both *power* and the return value.

## Example

```

INT i;
REAL(64) r, s;

r := 1.5L0;
s := RTL_Normalize_Real64_(r, i); ! s gets 0.75L0
                                   ! i gets 1

```

## Odd

The Odd function determines whether a number is even or odd. This function is not available in the native CRE library.

```
INT PROC RTL_Odd_Int32_( number );
    INT(32) number;                ! in TNS only
```

*number*

is the number that is examined to see if it is even or odd.

## Return Value

Odd returns

1 if *number* is odd  
0 if *number* is even

## Example

```
INT i;
INT(32) d := 17d;

i := RTL_Odd_Int32_(d);      ! i gets 1
i := RTL_Odd_Int32_(d + 1D); ! i gets 0
```

## Positive\_Diff

The Positive\_Diff functions return the arithmetic difference between two numbers if the first number is greater than the second. If the first number is less than the second number, the Positive\_Diff functions return zero. These functions are not available in the native CRE library.

```
INT      PROC RTL_Positive_Diff_Int16_( x, y );
    INT x, y;                                ! in   TNS only

INT(32)  PROC RTL_Positive_Diff_Int32_( x, y );
    INT(32) x, y;                            ! in   TNS only

INT(64)  PROC RTL_Positive_Diff_Int64_( x, y );
    INT(64) x, y;                            ! in   TNS only

REAL(32) PROC RTL_Positive_Diff_Real32_( x, y );
    REAL(32) x, y;                          ! in   TNS only

REAL(64) PROC RTL_Positive_Diff_Real64_( x, y );
    REAL(64) x, y;                          ! in   TNS only
```

$x$

is the minuend, the number from which the subtrahend is subtracted.

$y$

is the subtrahend, the number to subtract from the minuend.

## Return Value

The Positive\_Diff functions return:

$x - y$  if  $x > y$   
 0 if  $x$  is less than or equal to  $y$

|

## Example

```
INT i, j, k;

i := 17;
j := -18;
k := RTL_Positive_Diff_Int16_(i, j); ! k gets 35
k := RTL_Positive_Diff_Int16_(j, i); ! k gets 0
```

## Power

The Power functions raise a number to a specified power. These functions are not available in the native CRE library.

```

INT      PROC{CRE_Power_Int16_to_Int16_}(base, exponent);
           {RTL_Power_Int16_to_Int16_}
INT base, exponent;           ! in                      TNS only

INT(32)  PROC{CRE_Power_Int32_to_Int16_}(base, exponent);
           {RTL_Power_Int32_to_Int16_}
INT(32) base;                 ! in
INT exponent;                 ! in                      TNS only

INT(64)  PROC{CRE_Power_Int64_to_Int16_}(base, exponent);
           {RTL_Power_Int64_to_Int16_}
INT(64) base;                 ! in
INT exponent;                 ! in                      TNS only

REAL(32) PROC{CRE_Power_Real32_to_Int16_}(base, exponent);
           {RTL_Power_Real32_to_Int16_}
REAL(32) base;                ! in
INT exponent;                 ! in                      TNS only

REAL(32) PROC{CRE_Power_Real32_to_Real32_}(base, exponent);
           {RTL_Power_Real32_to_Real32_}
REAL(32) base;                ! in
REAL(32) exponent;            ! in                      TNS only

REAL(64) PROC{CRE_Power_Real64_to_Int16_}(base, exponent);
           {RTL_Power_Real64_to_Int16_}
REAL(64) base;                ! in
INT exponent;                 ! in                      TNS only

REAL(64) PROC{CRE_Power_Real64_to_Real64_}(base, exponent);
           {RTL_Power_Real64_to_Real64_}
REAL(64) base;                ! in
REAL(64) exponent;            ! in                      TNS only

```

*base*

is the number that is raised to a power.

*exponent*

is the exponent to which *base* is raised.

## Return Value

The Power functions return

$base^{exponent}$

## Considerations

The Power functions cause a fault if:

- $base = 0$  and  $exponent$  is less than or equal to 0
- $base < 0$  and  $exponent$  has a floating-point type
- An intermediate calculation causes an arithmetic overflow

## Example

```
INT i, j, k;
i := 2;
j := 3;
k := RTL_Power_Int16_to_Int16_(i, j); ! k gets 8
```

## Power2

The Power2 functions multiply a number by an integral power of 2. These functions are not available in the native CRE library.

```
REAL(64) PROC { CRE_Power2_Real64_ } ( base, exponent );
               { RTL_Power2_Real64_ }
    REAL(64) base;           ! in
    INT      exponent;       ! in
only                                     TNS
```

*base*

is the number to multiply by.

*exponent*

is an integral power of 2.

## Return Value

The Power2 functions return

$base * 2^{exponent}$

## Considerations

An arithmetic fault occurs if an overflow or underflow occurs in an intermediate calculation.

## Example

```
INT i;
REAL(64) r, s;

r := 7.0L0;
```



```
i := 3;
s := RTL_Power2_Real64_(r, i); ! s gets 56.0L0
```

## Random\_Set, Random\_Next

The `Random_Set` function sets a seed—the value of the initial random number—for a pseudo-random number generator. The `Random_Next` function returns the next pseudo-random number in the current sequence. These functions are not available in the native CRE library.

```
PROC CRE_Random_Set_( seed );
    INT seed;                ! in                TNS only
```

```
INT PROC CRE_Random_Next_;                ! TNS only
```

*seed*

is the beginning number of the pseudo-random number sequence.

## Considerations

- Each call to `CRE_Random_Set_` establishes a new seed value.
- Each call to `CRE_Random_Next_` returns the next pseudo-random number in the current sequence.

## Round

The `Round` functions return the nearest whole number to their parameter. These functions are not available in the native CRE library.

```
REAL(32) PROC RTL_Round_Real32_( number );
    REAL(32) number;                ! in                TNS only

REAL(64) PROC RTL_Round_Real64_( number );
    REAL(64) number;                ! in                TNS only
```

*number*

is the number to round.

## Return Value

The `Round` functions return:

```
truncate( number + 0.5 ) if number is greater than or equal to 0
truncate( number - 0.5 ) if number < 0
```

## Sign

The Sign functions return their first parameter with the sign adjusted according to the value specified in their second parameter. These functions are not available in the native CRE library.

INT	PROC RTL_Sign_Int16_( <i>number</i> , <i>sign</i> );	
INT( <i>number</i> , <i>sign</i> ;	! in	TNS only
INT(32)	PROC RTL_Sign_Int32_( <i>number</i> , <i>sign</i> );	
INT(32) <i>number</i> , <i>sign</i> ;	! in	TNS only
INT(64)	PROC RTL_Sign_Int64_( <i>number</i> , <i>sign</i> );	
INT(64) <i>number</i> , <i>sign</i> ;	! in	TNS only
REAL(32)	PROC RTL_Sign_Real32_( <i>number</i> , <i>sign</i> );	
REAL(32) <i>number</i> , <i>sign</i> ;	! in	TNS only
REAL(64)	PROC RTL_Sign_Real64_( <i>number</i> , <i>sign</i> );	
REAL(64) <i>number</i> , <i>sign</i> ;	! in	TNS only

*number*

is the number whose sign is modified.

*sign*

determines whether the Sign functions return a positive or negative number.

## Return Value

The Sign functions return:

absolute\_value(*number*) if *sign* is greater than or equal to 0  
 -absolute\_value(*number*) if *sign* < 0

## Example

```
INT i, j, k;

i := 17;
j := -18;
k := RTL_Sign_Int16_( i, j ); ! k gets -17
k := RTL_Sign_Int16_( j, i ); ! k gets 18
```

## Sin

The Sin functions return the sine of an angle.

REAL(32) PROC RTL_Sin_Real32_( <i>angle</i> ); REAL(32) <i>angle</i> ; ! in	TNS only
REAL(64) PROC RTL_Sin_Real64_( <i>angle</i> ); REAL(64) <i>angle</i> ; ! in	TNS only

*angle*

is the angle, expressed in radians, whose sine is returned.

### Return Value

The Sin functions return the sin of *angle*. The return value is in the range:

-1.0 less than or equal to sin(*angle*) less than or equal to 1.0

## Sinh

The Sinh functions return a hyperbolic sine.

REAL(32) PROC { CRE_Sinh_Real32_ RTL_Sinh_Real32_ } ( <i>number</i> ); REAL(32) <i>number</i> ; ! in	TNS only
REAL(64) PROC { CRE_Sinh_Real64_ RTL_Sinh_Real64_ } ( <i>number</i> ); REAL(64) <i>number</i> ; ! in	TNS only

*number*

is the number whose hyperbolic sine is returned.

### Return Value

The sinh functions return the hyperbolic sine of *number*. The return value is:

$$\frac{\exp( \textit{number} ) - \exp( -\textit{number} )}{2}$$

### Considerations

An arithmetic fault occurs if there is an overflow in any intermediate calculation.

## Split

The Split function separates a 64-bit floating-point number into integral and fractional parts.

```
REAL(64) PROC RTL_Split_Real64_(number, integral_part);
    REAL(64)      number;          ! in
    REAL(64) .EXT integral_part;  ! out                                TNS only
```

*number*

is the number to separate into an integral and fractional part.

*integral\_part*

is a reference parameter into which Split stores the integral part of *number*.

## Return Value

Split returns the fractional part of *number*.

## Example

```
REAL(64) r;
REAL(64) s;

r := RTL_Split_Real64_(1.6L0, s);      ! r gets 0.6L0
                                         ! s gets 1.0L0

r := RTL_Split_Real64_(-2.7L0, s);     ! r gets -0.7L0
                                         ! s gets -2.0L0
```

## Sqrt

The Sqrt functions return the square root of a number.

```
REAL(32) PROC { CRE_Sqrt_Real32_ } ( number );
               { RTL_Sqrt_Real32_ }
    REAL(32) number;                                ! TNS only

REAL(64) PROC { CRE_Sqrt_Real64_ } ( number );
               { RTL_Sqrt_Real64_ }
    REAL(64) number;                                ! TNS only
```

*number*

is the number whose square root is returned.

## Return Value

The Sqrt functions return the square root of *number*.

## Considerations

An arithmetic fault occurs if *number* is less than zero.

## Example

```
REAL(64) r, s;

r := 25.0L0;
s := RTL_Sqrt_Real64_(r); ! s gets 5.0L0
```

## Tan

The Tan functions return the tangent of an angle.

REAL(32) PROC RTL_Tan_Real32_( <i>angle</i> ); REAL(32) <i>angle</i> ; ! in	TNS only
REAL(64) PROC RTL_Tan_Real64_( <i>angle</i> ); REAL(64) <i>angle</i> ; ! in	TNS only

*angle*

is the angle, expressed in radians, whose tangent is returned.

## Return Value

The Tan functions return the tangent of *angle*:

$$\tan( \textit{angle} ) = \frac{\sin( \textit{angle} )}{\cos( \textit{angle} )}$$

## Considerations

$\tan(\textit{angle})$  is undefined if  $\cos(\textit{angle}) = 0$ .

## Tanh

The Tanh functions return a hyperbolic tangent.

REAL(32) PROC { CRE_Tanh_Real32_ RTL_Tanh_Real32_ } ( <i>number</i> ); REAL(32) <i>number</i> ; ! in	TNS only
REAL(64) PROC { CRE_Tanh_Real64_ RTL_Tanh_Real64_ } ( <i>number</i> ); REAL(64) <i>number</i> ; ! in	TNS only

*number*

is the number whose hyperbolic tangent is returned.

## Return Value

The Tanh functions return the hyperbolic tangent of *number*. The value returned is:

$$\tanh = \frac{\sinh(\textit{number})}{\cosh(\textit{number})}$$

## Considerations

An arithmetic fault occurs if an intermediate calculation causes an overflow.

## Truncate

The Truncate functions return the nonfractional part of a number. These functions are not available in the native CRE library.

REAL(32) PROC RTL_Truncate_Real32_( <i>number</i> );	
REAL(32) <i>number</i> ;	! in TNS only
REAL(64) PROC RTL_Truncate_Real64_( <i>number</i> );	
REAL(64) <i>number</i> ;	! in TNS only

*number*

is the number whose integral part is returned.

## Return Value

The Truncate functions return the number whose absolute value is the largest integer that does not exceed the absolute value of *number* and has the same sign as *number*.

The Truncate functions return:

`ceil(number)` if *number* < 0  
`modf(number)` if *number* is greater than or equal to 0

## Example

```
REAL(32) r;

r := RTL_Truncate_Real32_(1.6E0);      ! r gets 1
r := RTL_Truncate_Real32_(-2.7E0);     ! r gets -2
```

## Upper

The Upper function returns the smallest integer that is not less than *number*.

REAL(64) PROC RTL_Upper_Real64_( <i>number</i> );	
REAL(64) <i>number</i> ;	! in TNS only

*number*

is the number to which the Upper function is applied.

Return Value

Upper returns the smallest integer that is not less than *number*.

Example

```
REAL(64) r;

r := RTL_Upper_Real64_(1.8L0);      ! r gets 2.0L0
r := RTL_Upper_Real64_(-1.8L0);     ! r gets -1.0L0
```

# Sixty-Four-Bit Logical Operations (Bit Manipulation Functions)

This subsection describes the functions that manipulate 64-bit integer operands. [Table 7-3](#) on page 7-23 summarizes the 64-bit logical operations. Descriptions of the functions themselves follow.

Table 7-3. Sixty-Four-Bit Logical Operations

Function Name	Function Action
Shift_Left	Returns its 64-bit integer parameter logically shifted left.
Shift_Right	Returns its 64-bit integer parameter logically shifted right.
Complement	Returns the one's complement of its 64-bit integer parameter.
AND	Returns the bit-wise logical AND of its two 64-bit integer parameters.
OR	Returns the bit-wise logical OR of its two 64-bit integer parameters.
XOR	Returns the bit-wise logical XOR of its two 64-bit integer parameters.
Remainder	Returns the remainder from the integer division of its two 64-bit operands.

You can manipulate bits in 64-bit operands using the following functions.

```

INT(64) PROC RTL_Shift_Left_Int64_( bit_string, count );
  INT(64) bit_string;          ! in
  INT      count;              ! in
only                                     TNS

INT(64) PROC RTL_Shift_Right_Int64_( bit_string, count );
  INT(64) bit_string;          ! in
  INT      count;              ! in
only                                     TNS

INT(64) PROC RTL_Complement_Int64_( bit_string );
  INT(64) bit_string;          ! in
only                                     TNS

INT(64) PROC RTL_And_Int64_( bit_string, mask );
  INT(64) bit_string, mask;    ! in
only                                     TNS

INT(64) PROC RTL_Or_Int64_( bit_string, mask );
  INT(64) bit_string, mask;    ! in
only                                     TNS

INT(64) PROC RTL_Xor_Int64_( bit_string, mask );
  INT(64) bit_string, mask;    ! in
only                                     TNS

```

*bit\_string*

is a 64-bit bit string that is manipulated by the operator.

*mask*

is a sequence of bits that is applied to *bit\_string*.

*count*

specifies the number of bits to shift *bit\_string*.

## Return Value

The bit-manipulation functions return the value of *bit\_string* after applying the specified function—Shift\_Left, Shift\_Right, Complement, AND, OR, or XOR).

## Examples

```

INT(64) i,
  bits := %76543210F;
i := RTL_Shift_Left_Int64_(bits, 3);    ! I = %765432100F
i := RTL_Shift_Right_Int64_(bits, 3);   ! I = %007654321F
i := RTL_Complement_Int64_(bits);
                                         ! I = %17777777777777701234567F

```



```
i := RTL_And_Int64_(bits, %55555555F); ! I = %054541010F
i := RTL_Or_Int64_( bits, %55555555F); ! I = %077557755F
i := RTL_Xor_Int64_(bits, %55555555F); ! I = %023016745F
```

# Remainder

The Remainder function returns the 64-bit integer remainder from the integer division of its parameters.

```
INT(64) PROC RTL_Remainder_Int64_( dividend, divisor );
    INT(64) dividend, divisor; ! in
only
```

TNS

*dividend*  
is the 64-bit dividend.

*divisor*  
is the 64-bit divisor.

## Return Value

Remainder returns the remainder after dividing *dividend* by *divisor*.

# Decimal Conversion Functions

This subsection describes the decimal conversion functions. [Table 7-4](#) on page 7-25 lists the functions in this subsection.

**Table 7-4. Decimal Conversion Functions**

Function Name	Function Action
Decimal_to_Int	Converts a decimal number represented in ASCII to a 16-, 32-, or 64-bit integer.
Int_to_Decimal	Converts a 16-, 32-, or 64-bit integer to a decimal number represented in ASCII.

The native CRE library does not provide Decimal Conversion functions. If your program must perform these conversions, use language-specific run-time library decimal conversion functions.

## Decimal\_to\_Int

The `Decimal_to_Int` functions convert a string of decimal digits represented in ASCII characters to a binary value of type `INT`, `INT(32)`, or `INT(64)`. These functions are not available in the native CRE library.

```
INT PROC RTL_Decimal_to_Int16_( str, len, result );
  STRING .EXT str;      ! in
  INT    len;           ! in
  INT    .EXT result;    ! out                                TNS only

INT PROC RTL_Decimal_to_Int32_( str, len, result );
  STRING .EXT str;      ! in
  INT    len;           ! in
  INT(32) .EXT result;   ! out                                TNS only

INT PROC RTL_Decimal_to_Int64_( str, len, result );
  STRING .EXT str;      ! in
  INT    len;           ! in
  INT(64) .EXT result;   ! out                                TNS only
```

*str*  
contains the ASCII string to convert. It can have a maximum of 19 characters. The value is represented as a string of decimal digits in ASCII, possibly including a sign. [Table 7-5](#) on page 7-29 shows the representations for the sign.

*len*  
specifies the number of characters in *str*.

*result*  
contains the binary result. If *result* is not large enough to contain the converted result, *result* is undefined.

## Return Value

The `Decimal_to_Int` functions return one of the following values:

Return Value	Meaning
0	The operation was successful.
1	The size of the destination was not large enough to hold the value.
2	<i>len</i> was less than 1 or greater than 19.
3	<i>str</i> contained nonnumeric data.

## Examples

```
STRING .EXT s[0:4];;
INT    .EXT r[0:0];
```

```

INT          err;

s  ' := ' "1234";
err := RTL_Decimal_to_Int16_(s, 4, r);  ! R = 1234;

s  ' := ' %hB1323334;
err := RTL_Decimal_to_Int16_(s, 5, r);  ! R = -1234;

```

## Int\_to\_Decimal

The `Int_to_Decimal` functions convert an integer to a character string whose contents represent a decimal number. You can specify the format in which the sign of the number is stored in the character string. These functions are not available in the native CRE library.

```

INT PROC RTL_Int16_to_Decimal_( int16, str, len, sign_type );
    INT          int16;          ! in
    STRING .EXT  str;            ! out
    INT          len;            ! in
    INT          sign_type;      ! in                                TNS only

INT PROC RTL_Int32_to_Decimal_( int32, str, len, sign_type );
    INT(32)      int32;          ! in
    STRING .EXT  str;            ! out
    INT          len;            ! in
    INT          sign_type;      ! in                                TNS only

INT PROC RTL_Int64_to_Decimal_( int64, str, len, sign_type );
    INT(64)      int64;          ! in
    STRING .EXT  str;            ! out
    INT          len;            ! in
    INT          sign_type;      ! in                                TNS only

```

*int16*  
*int32*  
*int64*

contains the value to convert.

*str*

is a string of up to 19 bytes that contains the result of the conversion.

*len*

specifies the length of *str* in bytes.

*sign\_type*

specifies how to represent the sign in the result. [Table 7-5](#) on page 7-29 shows the literals you can use for *sign\_type* when you call `Int_to_Decimal`.

## Return Value

The Int\_to\_Decimal functions return one of the following values:

Return Value	Meaning
0	The operation was successful.
1	The size of the destination was not large enough to hold the value. Significant digits were truncated.
2	<i>len</i> was invalid. <i>len</i> must be greater than zero and less than or equal to 19.
3	You specified an unsigned DECIMAL result, but the source was a negative number. You need to specify a signed result.
4	You specified an invalid <i>sign-type</i> .

## Considerations

Each function in this group has a secondary entry point. If you invoke the function using the second entry point, the decimal string returned by the function is terminated with a single null character. The names of the secondary entry points are the same names as those given above except that a “c” is appended to the end of the word “decimal”:

Name as Shown Above	Name of Secondary Entry Point
RTL_Int16_to_Decimal_	RTL_Int16_to_Decimalc_
RTL_Int32_to_Decimal_	RTL_Int32_to_Decimalc_
RTL_Int64_to_Decimal_	RTL_Int64_to_Decimalc_

## Examples

```
STRING .EXT s[0:4];
INT    err;

err :=
    RTL_Int16_to_Decimal_( 1234, s, 5, RTL^Leading^separate );
    ! S = "+1234";

err :=
    RTL_Int16_to_Decimal_( -1234, s, 4, RTL^Trailing^embedded );
    ! S = %h313233B4;
```

**Table 7-5. Sign Types**

<b>Sign_type</b>	<b>Sign Storage</b>
RTL^Unsigned	A sign is not stored. The result is an unsigned string of digits.
RTL^Leading^embedded	The leftmost bit of the first byte is set to 1 to indicate a negative value.
RTL^Leading^separate	A “+” or “-” character occupies the first byte. “+” indicates a positive value, “-” indicates a negative value.
RTL^Trailing^embedded	The leftmost bit of the last byte is set to 1 for a negative value.
RTL^Trailing^separate	A “+” or “-” character occupies the last byte. “+” indicates a positive value, “-” indicates a negative value.



# String and Memory Block Functions

This section describes the string and memory block functions supported by the CRE.

The term “white space” in a function description includes spaces, newline characters, horizontal tabs, vertical tabs, and form feeds.

## String Functions

This subsection describes the string functions supported by the TNS CRE. Each function described in this section begins with the prefix `RTL_` or `CRE_`. Refer to [Using Standard Functions](#) on page 2-56 for more information.

**Table 8-1. TNS CRE String Functions** (page 1 of 2)

Function Name	Function Action
<a href="#">Atoi, Atol, Atof</a> on page 8-3	Converts a string of decimal characters to an integer or a real number.
<a href="#">Stcarg</a> on page 8-4	Scans a string until it finds one of the characters from another specified string.
<a href="#">Stccpy</a> on page 8-5	Copies characters from one string to another.
<a href="#">Stcd_I</a> on page 8-6	Converts a string of decimal characters to an integer.
<a href="#">Stcd_L</a> on page 8-7	Converts a string of decimal characters to a 32-bit integer.
<a href="#">Stch_I</a> on page 8-8	Converts a string of hexadecimal characters to an integer.
<a href="#">Stci_D</a> on page 8-9	Converts a signed integer to a string of decimal characters.
<a href="#">Stcpm</a> on page 8-10	Scans a string for a substring that matches a specified pattern.
<a href="#">Stcpma</a> on page 8-11	Determines whether a string starts with a substring that matches a specified pattern.
<a href="#">Stcu_D</a> on page 8-12	Converts an unsigned integer to a string of decimal characters.
<a href="#">Stpblk</a> on page 8-13	Scans a string for a non-white-space character.
<a href="#">Stpsym</a> on page 8-14	Gets a symbol from a string and stores it in another specified string.
<a href="#">Stptok</a> on page 8-15	Scans a string for a token and copies the token to another string.

**Table 8-1. TNS CRE String Functions** (page 2 of 2)

Function Name	Function Action
<a href="#">Strcat</a> on page 8-16	Concatenates two strings.
<a href="#">Strchr</a> on page 8-17	Scans a string for the first occurrence of a specified character.
<a href="#">Strcmp</a> on page 8-18	Compares two strings.
<a href="#">Strcpy</a> on page 8-19	Copies one string to another.
<a href="#">Strcspn</a> on page 8-20	Scans a string until it finds a character that is present in another specified string.
<a href="#">Strlen</a> on page 8-21	Returns the length of a string.
<a href="#">Strncat</a> on page 8-22	Concatenates one character string to the end of another.
<a href="#">Strncmp</a> on page 8-23	Compares two strings up to a specified maximum number of characters.
<a href="#">Strncpy</a> on page 8-24	Copies characters from one string to another.
<a href="#">Strpbrk</a> on page 8-25	Scans a string for the first occurrence of any character present in another string.
<a href="#">Strrchr</a> on page 8-26	Scans a string backwards for the last occurrence of a specified character.
<a href="#">Strspn</a> on page 8-27	Scans a string until it finds a character that is not in another specified string.
<a href="#">Strstr</a> on page 8-27	Determines whether one string is a substring of another string.
<a href="#">Strtod</a> on page 8-28	Converts a string of characters to a 64-bit floating-point number.
<a href="#">Strtol</a> on page 8-29	Converts a string of characters to a 32-bit integer using a specified base.
<a href="#">Strtoul</a> on page 8-30	Converts a string of characters to a 32-bit unsigned integer with a specified base.
<a href="#">Substring_Search</a> on page 8-32	Determine whether one string is a substring of another string.

[Table 8-2](#) on page 8-3 shows the string functions supported by the native CRE library. The native CRE does not provide pTAL prototypes for these functions. It provides only the function names. Note that the function names are case-sensitive. You can use the function prototypes in the C header files as examples when writing your own pTAL prototypes for these functions.



See the *Guardian Native C Library Calls Reference Manual* for descriptions of these functions.

**Table 8-2. Native CRE String Functions**

Function Name	Native CRE Library Function Name
strcat	Concatenates two strings.
strchr	Scans a string for the first occurrence of a specified character.
strcmp	Compares two strings.
strcpy	Copies one string to another.
strcspn	Scans a string until it finds a character that is present in another specified string.
strlen	Returns the length of a string.
strncat	Concatenates one character string to the end of another.
strncmp	Compares two strings up to a specified maximum number of characters.
strncpy	Copies characters from one string to another.
strpbrk	Scans a string for the first occurrence of any character present in another string.
strrchr	Scans a string backwards for the last occurrence of a specified character.
strspn	Scans a string until it finds a character that is not in another specified string.
strstr	Determines whether one string is a substring of another string.
strtok	Scans a string for a token and copies the token to another string.

## Atoi, Atol, Atof

The Atoi, Atol, and Atof functions convert a string of decimal characters to an integer or floating-point number. These functions are not available in the native CRE library.

```

INT      PROC RTL_Atoi_( str );
  STRING .EXT str;           ! in   TNS only

INT(32)  PROC RTL_Atoll_( str );
  STRING .EXT str;           ! in   TNS only

REAL(64) PROC RTL_Atof_( str );
  STRING .EXT str;           ! in   TNS only

```

*str*

is a pointer to the string to convert.

## Return Value

The Atoi, Atol, and Atof functions return the converted value of *str*. They return zero if the first non-white-space character in *str* is not a sign character (“+” or “-”), a decimal digit, or, for Atof, a decimal point.

## Considerations

- The Atoi, Atol, and Atof functions skip leading white-space characters in *str*.
- The first non-white-space character can be optionally a plus sign or a minus sign.
- For Atof only, the first non-white-space character can be a decimal point.
- The functions convert successive characters until they encounter either a non-decimal character or a zero (null) byte.

## Example

```
STRING .s[0:17] := [" 275",0];
INT i;
STRING .t[0:17] := [" 275.9E2",0];
REAL(64) r;

i := RTL_Atoi_(s);  ! i gets 275
r := RTL_Atof_(t);  ! r gets 27590
```

## Stcarg

The Stcarg functions scan a character string until any character that appears in a second string is encountered in the first string. These functions are not available in the native CRE library.

```
INT      PROC RTL_Stcarg_( str, stop_chars );
  STRING .str;           ! in
  STRING .stop_chars;    ! in                      TNS only

INT(32)  PROC RTL_StcargX_( str, stop_chars );
  STRING .EXT str;       ! in
  STRING .EXT stop_chars; ! in                      TNS only
```

*str*

is a pointer to the string to scan.

*stop\_chars*

is a pointer to a string containing the characters to scan for in *str*.

## Return Value

The Stcarg functions return the number of bytes scanned at the beginning of *str* before a character in *stop\_chars* was encountered. If no characters from *stop\_chars* are found in *str*, the Stcarg functions return the length of *str*.

## Considerations

- The Stcarg functions scan *str* until any character in *stop\_chars* is found in *str* or until they encounter a zero (null) byte in *str*.
- The Stcarg functions ignore text in *str* that is inside matched single or double quotes or that follows an unmatched single or double quote. They also ignore any character that is preceded by a backslash.
- Both *str* and *stop\_chars* must be terminated by a zero (null) byte to stop the scan.

## Example

```
INT count;
STRING .s[0:7] := ["ABCCDEF",0];
STRING .tar[0:3] := ["abcd",0];

count := RTL_Stcarg_(s, tar); ! count gets 3
```

## Stccpy

The Stccpy functions copy not more than a specified number of characters from one string to another. These functions are not available in the native CRE library.

```
INT PROC RTL_Stccpy_( dest, source, max_bytes );
  STRING .dest;           ! out
  STRING .source;         ! in
  INT     max_bytes;       ! in                               TNS only

INT(32) PROC RTL_StccpyX_( dest, source, max_bytes );
  STRING .EXT dest;       ! out
  STRING .EXT source;     ! in
  INT(32) max_bytes;      ! in                               TNS only
```

*dest*

is a pointer to the destination of the copy.

*source*

is a pointer to the string to copy.

*max\_bytes*

specifies the maximum number of bytes to copy. *max\_bytes* must be a positive number.

## Return Value

The Stccpy functions return the number of characters copied from *source* to *dest*.

## Considerations

- *source* must be terminated by a zero (null) byte to stop the scan.
- The Stccpy functions terminate *dest* with a null byte, independent of whether the maximum count or a null byte in *source* stopped the copy operation.

## Example

```
INT i;
STRING .des[0:7];
STRING .src[0:20] := ["abc",0,"def"];

i := RTL_Stccpy_(des, src, 8); ! i gets 3, des gets ["abc",0]
```

## Stcd\_I

The Stcd\_I functions convert a string of decimal characters to a 16-bit integer. These functions are not available in the native CRE library.

```
INT PROC RTL_Stcd_I_( str, integer );
  STRING .str;           ! in
  INT    .integer;       ! out           TNS only

INT PROC RTL_Stcd_IX_( str, integer );
  STRING .EXT str;       ! in
  INT    .EXT integer;   ! out           TNS only
```

*str*

is a pointer to the string to convert.

*integer*

is a pointer to the resulting integer.

## Return Value

The Stcd\_I functions return the number of characters scanned.

## Considerations

- The string to convert can contain an optional sign before the string of digits.
- The Stcd\_I functions do not skip leading white space.
- The Stcd\_I functions stop scanning if they encounter a non-decimal character or a zero (null) byte.

## Example

```
INT count;
STRING .s[0:7] := ["17xy",0];
INT      i;

count := RTL_Stcd_I_(s, i); ! count gets 2, i gets 17
```

## Stcd\_L

The Stcd\_L functions convert a string of decimal characters to a 32-bit integer. These functions are not available in the native CRE library.

```
INT PROC RTL_Stcd_L_( str, longint );
  STRING      .str;          ! in
  INT(32)     .longint;      ! out          TNS only

INT PROC RTL_Stcd_LX_( str, longint );
  STRING      .EXT str;      ! in
  INT(32)     .EXT longint;   ! out          TNS only
```

*str*

is a pointer to the string to convert.

*longint*

is a pointer to the resulting 32-bit integer.

## Return Value

The Stcd\_L functions return the number of characters scanned.

## Considerations

- *str* can contain an optional sign followed by a string of digits.
- The Stcd\_L functions do not skip leading white space.
- The Stcd\_L functions stop scanning if they encounter a non-decimal character or a zero (null) byte.

## Example

```
INT count;
STRING .s[0:17] := ["128456xq7",0];
INT(32) d;
```

```
count := RTL_Stcd_L_(s, d); ! count gets 6, d gets 128456
```

## Stch\_I

The Stch\_I functions convert a string of hexadecimal characters to an integer. These functions are not available in the native CRE library.

<pre>INT PROC RTL_Stch_I_( str, integer );   STRING .str;           ! in   INT    .integer;        ! out                                 TNS only  INT PROC RTL_Stch_IX_( str, integer );   STRING .EXT str;        ! in   INT    .EXT integer;    ! out                                 TNS only</pre>
---

*str*

is a pointer to the string to convert.

*integer*

is a pointer to the resulting integer.

## Return Value

The Stch\_I functions return the number of characters scanned, or zero if *str* does not begin with a valid hexadecimal character.

## Considerations

- The end of *str* is defined by the location of a zero (null) byte.
- The Stch\_I functions do not skip leading white space.
- The Stch\_I functions stop scanning if they encounter a non-hexadecimal character.

## Example

```
INT count;
STRING .s[0:7] := ["1a2xy",0];
INT    i;
```

```
count := RTL_Stch_I_(s, i); ! count gets 3, i gets 418.
```

## Stci\_D

The Stci\_D functions convert a signed integer to a string of decimal characters. These functions are not available in the native CRE library.

```

INT PROC RTL_Stci_D_( str, integer, max_bytes );
  STRING .str;                ! out
  INT    integer;              ! in
  INT    max_bytes;            ! in                                TNS only

INT PROC RTL_Stci_DX_( str, integer, max_bytes );
  STRING .EXT str;            ! out
  INT    integer;              ! in
  INT    max_bytes;            ! in                                TNS only

```

*str*

is a pointer to the converted decimal string.

*integer*

is the integer to convert.

*max\_bytes*

is the size of the buffer, *str*, provided for the conversion.

## Return Value

The Stci\_D functions return the length of the resulting string, excluding the terminating null character.

## Considerations

- The Stci\_D functions do not store leading zeros in the output string.
- The output string is terminated with a null byte.
- If *integer* is negative, the output string is preceded by a minus sign.
- If *integer* is zero, a single zero character is stored.
- The buffer at *str* must be large enough to hold a minus sign, if *integer* is negative, plus the converted number, and a null byte.

If there are more than *max\_bytes* to store, the Stci\_D functions store the minus sign if *integer* is negative, followed by as many bytes as will fit while still storing a null character at the end of *str*.

If all digits are not stored, the digits stored are the least significant digits of the converted integer.

<i>Integer Negative?</i>	<i>Max Bytes</i>	<i>Max Digits Stored</i>
No	N	N - 1
Yes	N	N - 2

### Example

```
INT count;
STRING .s[0:7];
INT    i := 17;

! count gets 2, s gets ["17",0]

count := RTL_Stci_D_(s, i, 8);
```

### Stcpm

The Stcpm functions scan a string for the first occurrence of a substring that matches a specified pattern. This function is not available in the native CRE library.

```
INT PROC RTL_Stcpm_( str, pat, match );
  STRING .str;           ! in
  STRING .pat;           ! in
  INT    .match;         ! out           TNS only

INT PROC RTL_StcpmX_( str, pat, match );
  STRING .EXT str;       ! in
  STRING .EXT pat;       ! in
  INT(32) .EXT match;    ! out           TNS only
```

*str*  
is a pointer to the string to scan.

*pat*  
is a pointer to the pattern string. The pattern is specified using regular expression notation:

- ? matches any character.
- s\* matches zero or more occurrences of *s*.
- s+ matches one or more occurrences of *s*.
- s matches *s*.

*match*  
is a pointer, returned by Stcpm, to the substring in *str* that matches *pat*.



## Return Value

The Stcpm functions return the length of the first matching substring, if a match was found. If a match was not found, the functions returns -1.

## Considerations

- The ends of *str* and *pat* are defined by the location of a zero (null) byte.
- You can use a backslash (\) as an escape character if you need to match a special character (?, \*, or +).
- Note that *match* must be a pointer to a character pointer.

## Example

```
STRING .s[0:17] := ["xxFORTUNE 500",0];
STRING .pattern[0:4] := ["FO?T",0];
STRING .match;
INT i;

! i get 4.
! @match gets @s[2]

i := RTL_Stcpm_(s, pattern, @match);
```

## Stcpma

The Stcpma functions scan a string to determine whether it starts with a substring that matches a specified pattern. These functions are not available in the native CRE library.

```
INT PROC RTL_Stcpma_( str, pat );
    STRING .str;           ! in
    STRING .pat;           ! in      TNS only

INT PROC RTL_StcpmaX_( str, pat );
    STRING .EXT str;       ! in
    STRING .EXT pat;       ! in      TNS only
```

*str*

is a pointer to the string to scan.

*pat*

is a pointer to the pattern string. The pattern is specified using regular expression notation:

- ? matches any character.
- s*\* matches zero or more occurrences of *s*.
- s*+ matches one or more occurrences of *s*.
- s* matches *s*.

## Return Value

The Stcpma functions return the length of the matching substring, if found. If a matching substring is not found, they return -1.

## Considerations

- The ends of *str* and *pat* are defined by the location of a zero (null) byte.
- You can use a backslash (\) as an escape character if you need to match a special character.

## Example

```
STRING .s[0:17] := ["FORTUNE 500",0];
STRING .pattern[0:4] := ["FO?T",0];
INT      i;

i := RTL_Stcpma_(s, pattern); ! i gets 4
```

## Stcu\_D

The Stcu\_D functions convert an unsigned integer to a string of decimal characters. These functions are not available in the native CRE library.

```
INT PROC RTL_Stcu_D_( str, integer, max_bytes );
  STRING .str;           ! out
  INT     integer;        ! in
  INT     max_bytes;      ! in
                                     TNS only

INT PROC RTL_Stcu_DX_( str, integer, max_bytes );
  STRING .EXT str;        ! out
  INT     integer;        ! in
  INT     max_bytes;      ! in
                                     TNS only
```

*str*

is a pointer to the converted decimal string.

*integer*

is the integer to convert.

*max\_bytes*

is the number of bytes allocated at *str* for the conversion.

## Return Value

The Stcu\_D functions return the length of the resulting string, excluding the terminating null character.

## Considerations

- The Stcu\_D functions do not produce leading zeros in the output string.
- If *integer* is negative, the output string is preceded by a minus sign.
- *str* is terminated with a null byte.
- If *integer* is zero, a single zero character is produced.
- The buffer at *str* must be large enough to hold the converted number and a null byte. If there are more than *max\_bytes* minus one digits in the converted number, the Stcu\_D functions store *max\_bytes* minus one digits and a null character. The digits stored are the least significant digits of the converted integer.

## Example

```
INT count;
STRING .s[0:7];
INT    i := %100002;

! count gets 5, s gets ["32770",0]

count := RTL_Stcu_D(s, i, 8);
```

## Stpblk

The Stpblk functions scan a string for a non-white-space character. These functions are not available in the native CRE library.

```
INT PROC RTL_Stpblk_( str );
    STRING .str;           ! in           TNS only

INT(32) PROC RTL_StpblkX_( str );
    STRING .EXT str;       ! in           TNS only
```

*str*

is a pointer to the string to scan.

## Return Value

The Stpblk functions return the address of the first non-white-space character in *str*.

## Considerations

The end of *str* is defined by the location of a zero (null) byte.

## Example

```
STRING .s[0:17] := ["  Skip Spaces",0];
STRING .start;

!@start gets @s[2]

@start := RTL_Stpblk_(s);
```

## Stpsym

The Stpsym functions copy a symbol from one string to another string. A symbol consists of an alphabetic character followed by zero or more alphanumeric characters. These functions are not available in the native CRE library.

```
INT PROC RTL_Stpsym_( source, sym, symlen );
  STRING .source;           ! in
  STRING .sym;              ! out
  INT     symlen;           ! in
                                TNS only

INT(32) PROC RTL_StpsymX_( source, sym, symlen );
  STRING .EXT source;       ! in
  STRING .EXT sym;          ! out
  INT     symlen;           ! in
                                TNS only
```

*source*

is a pointer to the string to scan.

*sym*

is a pointer, on return, to the symbol.

*symlen*

specifies the number of bytes allocated in *sym* for the symbol.

## Return Value

The Stpsym functions return the address of the next character in *str* after the symbol. If a symbol is not found, the address of the start of *str*, *@str* is returned.

## Considerations

- A symbol consists of an alphabetic character followed by zero or more alphanumeric characters and terminated by a blank character.
- White space is not skipped.
- The end of *str* is defined by the location of a zero (null) byte.
- To ensure space for a null byte after the symbol, the value you pass in *symlen* must be at least one greater than the longest symbol that can be scanned.

## Example

```
STRING .s[0:17] := ["ident1 ident2",0];
STRING .symbol[0:9];
STRING .next;

!@next gets @s[6]
!symbol gets ["ident1", 0]

@next := RTL_Stpsym_(s, symbol, 10);
```

## Stptok

The Stptok functions scan for the next token in a string. A token consists of all characters from the beginning of the string up to, but not including, any character that appears in a second string, a string of delimiter characters. These functions are not available in the native CRE library.

```
INT PROC RTL_Stptok_( str, token, tokenlen, stop_chars );
    STRING .str;           ! in
    STRING .token;         ! out
    INT     tokenlen;       ! in
    STRING .stop_chars;     ! in           TNS only

INT(32) PROC RTL_StptokX_( str, token, tokenlen, stop_chars
);
    STRING .EXT str;       ! in
    STRING .EXT token;     ! out
    INT     tokenlen;      ! in
    STRING .EXT stop_chars; ! in           TNS only
```

*str*

is a pointer to the string to scan.

*token*

is a pointer to the string to which the token is copied.

*tokenlen*

specifies the number of bytes allocated in *token* for the token. Tokens greater than this length are truncated.

*stop\_chars*

is a pointer to a string that consists of token separator characters; that is, characters that are not part of the token.

## Return Value

The Stptok functions return the address of the next character in *str* after the token (the delimiter that stopped the scan).

## Considerations

- Both *str* and *stop\_chars* must be terminated by a zero (null) byte to stop the scan.
- White space is not skipped at the beginning of *str*.
- A null byte in *token* immediately following the token.

## Example

```
STRING .s[0:17] := ["Token1, token2",0];
STRING .token[0:9];
STRING .tar[0:3] := [",-;",0];
STRING .next;
```

```
!@next gets @s[6]
!token gets ["Token1", 0]
```

```
@next := RTL_Stptok_(s, token, 10, tar);
```

## Strcat

The Strcat functions concatenate two strings.

```
INT PROC RTL_Strcat_( first, second );
  STRING .first;           ! in/out
  STRING .second;          ! in           TNS only

INT(32) PROC RTL_StrcatX_( first, second );
  STRING .EXT first;       ! in/out
  STRING .EXT second;      ! in           TNS only
```

*first*

is a pointer to the string to which the second string is concatenated.

*second*

is a pointer to the string that is concatenated to the first string.

## Return Value

The Strcat functions return the address of the resulting string, *@first*.

## Considerations

- Both *first* and *second* must be terminated by a zero (null) byte to stop the scan.
- You must ensure that there are enough bytes at the end of *first* to accommodate the bytes in *second*.
- The Strcat functions append a null byte to the result.

## Example

```
STRING fir[0:100] := ["Front",0];
STRING .sec[0:100] := ["Back",0];
STRING .temp;
```

```
! @temp gets @fir
! fir gets ["FrontBack",0]
```

```
@temp := RTL_Strcat_(fir, sec);
```

## Strchr

The Strchr functions scan a string for the first occurrence of a specified character.

<pre>INT PROC RTL_Strchr_( str, char );   STRING .str;           ! in   INT     char;           ! in           TNS only</pre>			
<pre>INT(32) PROC RTL_StrchrX_( str, char );   STRING .EXT str;       ! in   INT     char;          ! in           TNS only</pre>			

*str*

is a pointer to the string to scan.

*char*

is the character to search for. Only bits <8:15> of *char* are used for the search character.

## Return Value

The Strchr functions return the address of the first occurrence in *str* of *char*. If *str* does not contain *char*, zero is returned.

## Considerations

*str* must be terminated by a zero (null) byte to stop the scan.

## Example

```
STRING .s[0:100] := ["Find me",0];
STRING .char_ptr;

! @char_ptr gets @s[5]

@char_ptr := RTL_Strchr_(s, "m");
```

## Strcmp

The Strcmp functions compare two strings.

INT PROC RTL_Strcmp_( <i>str1</i> , <i>str2</i> );	
STRING . <i>str1</i> ;	! in
STRING . <i>str2</i> ;	! in TNS only
INT PROC RTL_StrcmpX_( <i>str1</i> , <i>str2</i> );	
STRING .EXT <i>str1</i> ;	! in
STRING .EXT <i>str2</i> ;	! inv TNS only

*str1*

is a pointer to the first string.

*str2*

is a pointer to the second string.

## Return Value

The Strcmp functions return a value that is

```
< 0 if str1 < str2
  0 if str1 = str2
> 0 if str1 > str2
```

## Considerations

- Both *str1* and *str2* must be terminated by a zero (null) byte to stop the scan.
- For less-than and greater-than comparisons, characters in *str1* and *str2* are compared using the characters' numeric values in the ASCII collating sequence.



- If *str1* and *str2* have different lengths but match to the end of the shorter string, the shorter string is defined to be less than the longer string.

## Example

```
INT sign;
STRING .s1[0:100] := ["Opus is a penguin",0];
STRING .s2[0:100] := ["Opus is a flightless water fowl",0];

sign := RTL_Strcmp_(s1, s2); ! sign gets a positive number
```

## Strcpy

The Strcpy functions copy one string to another.

<pre>INT PROC RTL_Strcpy_( dest, source );   STRING .dest;           ! out   STRING .source;          ! in                                 TNS only  INT(32) PROC RTL_StrcpyX_( dest, source );   STRING .EXT dest;        ! out   STRING .EXT source;      ! in                                 TNS only</pre>
---

*dest*

is a pointer to the destination string.

*source*

is a pointer to the string to copy.

## Return Value

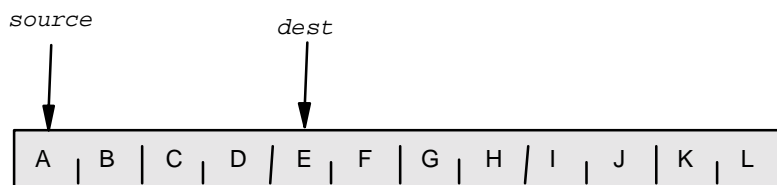
The Strcpy functions return the address of the resulting string, @*dest*.

## Considerations

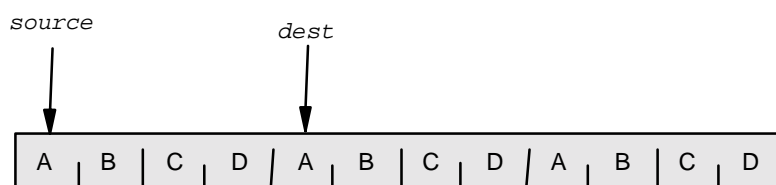
- *source* must be terminated by a zero (null) byte to stop the scan.
- If the strings overlap, data is not preserved. That is, if *dest* begins within *source*, the string at *dest* might not be a copy of what was stored at *source*. For example, [Figure 8-1](#) and [Figure 8-2](#) on page 8-20 show the before image and after image from the following code sequence:

```
STRING .s[0:11];
STRING .d;

s ' := ' "ABCDEFGHijkl";
@d := @s[4];
d ' := ' s FOR 12 BYTES;
```

**Figure 8-1. Strings in Memory Before Copying Source to Destination**

VST 801.VSD

**Figure 8-2. Strings in Memory After Copying Source to Destination**

VST 802.VSD

## Example

```
STRING .des[0:10];
STRING .src[0:100] := ["Example", 0];

CALL RTL_Strcpy_(des, src); ! des gets ["Example",0]
```

## Strcspn

The Strcspn functions scan a string until they find a character that is found in another string. These functions are not available in the native CRE library.

```
INT PROC RTL_Strcspn_( str, stop_chars );
    STRING .str;           ! in
    STRING .stop_chars;    ! in           TNS only

INT(32) PROC RTL_StrcspnX_( str, stop_chars );
    STRING .EXT str;       ! in
    STRING .EXT stop_chars; ! in           TNS only
```

*str*

is a pointer to the string to scan.

*stop\_chars*

is a pointer to a string containing the characters to scan for in *str*.

## Return Value

The Strcspn functions return the number of bytes scanned at the beginning of *str* before a character in *stop\_chars* was encountered. If no characters from *stop\_chars* are found in *str*, the length of *str* is returned.

## Considerations

Both *str* and *stop\_chars* must be terminated by a zero (null) byte to stop the scan.

## Example

```
INT count;
STRING .s[0:19] := ["Next token, please",0];
STRING .tar[0:3] := [";,:",0];

count := RTL_Strcspn_(s, tar); ! count gets 10
```

## Strlen

The Strlen functions return the length of a string.

INT PROC RTL_Strlen_( <i>str</i> );	
STRING . <i>str</i> ;	! in TNS only
INT(32) PROC RTL_StrlenX_( <i>str</i> );	
STRING .EXT <i>str</i> ;	! in TNS only

*str*

is a pointer to the string.

## Return Value

The Strlen functions return the length of *str* in bytes.

## Considerations

- *str* must be terminated by a zero (null) byte to stop the scan.
- The value returned does not include the null character at the end of *str*.

## Example

```
INT i;
STRING .s[0:7] := ["1234567",0];
```

```
i := RTL_Strlen_(s); ! i gets 7
```

## Strncat

The Strncat functions concatenate not more than a specified number of characters from one string to another.

```
INT PROC RTL_Strncat_( first, second, max_bytes );
  STRING .first;           ! out
  STRING .second;          ! in
  INT     max_bytes;        ! in
only                                     TNS

INT(32) PROC RTL_StrncatX_( first, second, max_bytes );
  STRING .EXT first;        ! out
  STRING .EXT second;       ! in
  INT(32) max_bytes;        ! in
only                                     TNS
```

*first*

is a pointer to the string to which the second string is concatenated.

*second*

is a pointer to the string that is concatenated to the first string.

*max\_bytes*

specifies the maximum number of characters from *second* to concatenate to *first*. *max\_bytes* must be a positive number.

## Return Value

The Strncat functions return the address of the resulting string, *@first*.

## Considerations

- Both *first* and *second* must be terminated by a zero (null) byte to stop the scan.
- You must ensure that there are enough bytes at the end of *first* to accommodate *max\_bytes* additional bytes.
- The functions append a null byte to the result.

## Example

```
STRING .fir[0:100] := ["Front",0];
STRING .sec[0:100] := ["Back",0];
STRING .temp;
! @temp gets @fir
```

```
! fir gets ["FrontBa",0]

@temp := RTL_Strncat_(fir, sec, 2);
```

## Strncmp

The Strncmp functions compare two strings for a specified maximum number of characters.

```
INT PROC RTL_Strncmp_( str1, str2, max_bytes );
    STRING .str1;           ! in
    STRING .str2;           ! in
    INT     max_bytes;       ! in                               TNS only

INT PROC RTL_StrncmpX_( str1, str2, max_bytes );
    STRING .EXT str1;       ! in
    STRING .EXT str2;       ! in
    INT(32) max_bytes;      ! in                               TNS only
```

*str1*

is a pointer to the first string.

*str2*

is a pointer to the second string.

*max\_bytes*

specifies the maximum number of characters to compare. *max\_bytes* must be a positive number.

## Return Value

Strncmp returns a value that is

```
< 0 if str1 < str2
  0 if str1 = str2
> 0 if str1 > str2
```

## Considerations

- Both *str1* and *str2* must be terminated by a zero (null) byte to stop the scan.
- For less-than and greater-than comparisons, characters in *str1* and *str2* are compared using the characters' values in the ASCII collating sequence.
- If one of the strings has fewer than *max\_bytes* characters and matches the beginning of the other string, the shorter string is defined to be less than the longer string, even if the length of the longer string is greater than the length specified by *max\_bytes*.

- If *max\_bytes* is greater than the length of both strings and if the two strings have different lengths but match to the end of the shorter string, the shorter string is defined to be less than the longer string.

## Example

```
INT sign;
STRING .s1[0:100] := ["Opus is a penguin",0];
STRING .s2[0:100] := ["Opus is a flightless water fowl",0];

sign := RTL_Strncmp_(s1, s2, 10); ! sign gets 0
```

## Strncpy

The Strncpy functions copy not more than a specified number of characters from one string to another.

```
INT PROC RTL_Strncpy_( dest, source, max_bytes );
  STRING .dest;           ! out
  STRING .source;         ! in
  INT     max_bytes;       ! in
only                                     TNS

INT(32) PROC RTL_StrncpyX_( dest, source, max_bytes );
  STRING .EXT dest;       ! out
  STRING .EXT source;     ! in
  INT(32) max_bytes;      ! in
only                                     TNS
```

*dest*

is a pointer to the destination string.

*source*

is a pointer to the string to copy.

*max\_bytes*

specifies the maximum number of characters to copy; *max\_bytes* must be a positive number.

## Return Value

The Strncpy functions return the address of the resulting string, @*dest*.

## Considerations

- The end of *source* is defined by the location of a zero (null) byte.

- If the length of *source* is greater than *max\_bytes*, the functions do not store a null character.
- The Strncpy functions copy bytes from *source* to *dest* until *max\_bytes* have been copied or a null byte is encountered in *source*:
  - If *max\_bytes* are copied without encountering a null byte in *source*, does a null byte is not stored in *dest*.
  - If a null byte is encountered in *source* before *max\_bytes* are copied, the functions copy the null byte and pad *dest* with additional null bytes until a total of *max\_bytes* have been written in *dest*.

## Example

```
STRING .des[0:100];
STRING .src[0:100] := ["HP Computers Incorporated",0];

CALL RTL_Strncpy_(des, src, 6); ! des gets ["HP",0];
```

## Strpbrk

The Strpbrk functions scan a string until they find any character in the string that also appears in a second string.

<pre>INT PROC RTL_Strpbrk_( str, stop_chars );   STRING .str;           ! in   STRING .stop_chars;     ! in                            TNS only  INT(32) PROC RTL_StrpbrkX_( str, stop_chars );   STRING .EXT str;        ! in   STRING .EXT stop_chars; ! in                            TNS only</pre>
---

*str*

is a pointer to the string to scan.

*stop\_chars*

is a pointer to the string containing the characters to scan for in *str*.

## Return Value

The Strpbrk functions return the address of the first occurrence in *str* of any character in *stop\_chars*. If *str* does not contain any of the characters in *stop\_chars*, zero is returned.

## Considerations

Both *str* and *stop\_chars* must be terminated by a zero (null) byte to stop the scan.

## Example

```
STRING .s[0:100] := ["800-555-1212",0];
STRING .tar[0:100] := ["-;,.",0];
STRING .char_ptr;
```

```
! @char_ptr gets @s[3]
```

```
@char_ptr := RTL_Strpbrk_(s, tar);
```

## Strchr

The Strchr functions scan a string backwards for the last occurrence of a specified character.

INT PROC RTL_Strrchr_( <i>str</i> , <i>char</i> );	
STRING . <i>str</i> ;	! in
INT <i>char</i> ;	! in                      TNS only
INT(32) PROC RTL_StrrchrX_( <i>str</i> , <i>char</i> );	
STRING .EXT <i>str</i> ;	! in
INT <i>char</i> ;	! in                      TNS only

*str*

is a pointer to the string to scan.

*char*

is the character to search for. Only bits <8:15> of *char* are used for the search character.

## Return Value

The Strchr functions return the address of the last occurrence in *str* of *char*. If *str* does not contain *char*, zero is returned.

## Considerations

*str* must be terminated by a zero (null) byte to stop the scan. The terminating zero byte must be at the right end of the buffer.

## Example

```
STRING .s[0:100] := ["This is an example",0];
STRING .char_ptr;
```

```
! @char_ptr gets @s[13]
```

```
@char_ptr := RTL_Strrchr_(s, "a");
```



## Strspn

The Strspn functions scan a string until they find a character in the string that does not appear in a second string.

```

INT PROC RTL_Strspn_( str, span_chars );
  STRING .str;                ! in
  STRING .span_chars;          ! in                      TNS only

INT(32) PROC RTL_StrspnX_( str, span_chars );
  STRING .EXT str;              ! in
  STRING .EXT span_chars;        ! in                      TNS only

```

*str*

is a pointer to the string to scan.

*span\_chars*

is a pointer to the string containing the characters to scan for in *str*.

## Return Value

The Strspn functions return the number of bytes scanned at the beginning of *str* before a character is encountered that is not in *span\_chars*. If all characters in *str* appear in *span\_chars*, the length of *str* is returned.

## Considerations

Both *str* and *span\_chars* must be terminated by a zero (null) byte to stop the scan.

## Example

```

INT count;
STRING .s[0:100] := ["An example: how to use RTL_STRSPN",0];
STRING .tar[0:100]:= ["ABCDEFGHIJKLMNOPQRSTUVWXYZ" &
                      "abcdefghijklmnopqrstuvwxyz",0];
count := RTL_Strspn_(s, tar); ! count gets 10

```

## Strstr

The Strstr functions determine whether one string is a substring of a second string.

```

INT PROC RTL_Strstr_( str, substr );
  STRING .str;                ! in
  STRING .substr;              ! in                      TNS only

INT(32) PROC RTL_StrstrX_( str, substr );
  STRING .EXT str;              ! in
  STRING .EXT substr;            ! in                      TNS only

```

*str*

is a pointer to the string to scan.

*substr*

is a pointer to the substring to scan for in *str*.

## Return Value

The Strstr functions return:

- The address of *substring* within *str* if *substr* is found.
- Zero if *substr* is not found within *str*.

## Considerations

Both *str* and *substr* must be terminated by a zero (null) byte to stop the scan.

## Example

```
STRING .s[0:17] := ["FORTUNE 500",0];
STRING .subs[0:4] := ["500",0];
STRING .subptr;
```

```
@subptr := RTL_Strstr_(s, subs); ! @subptr gets @s[8]
```

## Strtod

The Strtod functions convert a string of characters to a 64-bit floating-point number.

REAL(64)	PROC CRE_Strtod_(	<i>str</i> ,	<i>end_scan</i>	)	
STRING	. <i>str</i> ;		!	in	
INT	. <i>end_scan</i> ;		!	out	TNS only
REAL(64)	PROC CRE_StrtodX_(	<i>str</i> ,	<i>end_scan</i>	)	
STRING	.EXT <i>str</i> ;		!	in	
INT(32)	.EXT <i>end_scan</i> ;		!	out	TNS only

*str*

is a pointer to the string to convert.

*end\_scan*

is the address in *str* where the conversion ended. The pointer returned through *end\_scan* points to the first unrecognized character in *str*.

## Return Value

The Strtod functions return the converted value.

## Considerations

- *str* must be terminated by a zero (null) byte to stop the scan.
- Leading white space is skipped.
- *str* is a string of digits, optionally preceded by a sign.
- The functions stop scanning if they encounter an unrecognized character.

## Example

```
STRING .s[0:7] := ["52431",0];
STRING .scanend;
REAL(64) d;

! d gets 52431.
! @scanend gets @s[5]

d := CRE_Strtod_(s, @scanend );
```

## Strtol

The Strtol functions convert a string of characters to a 32-bit integer using a specified base.

```
INT(32) PROC CRE_Strtol_( str, end_scan, base );
    STRING .str;           ! in
    INT     .end_scan;      ! out
    INT     base;           ! in
                                TNS only

INT(32) PROC CRE_StrtolX_( str, end_scan, base );
    STRING .EXT str;        ! in
    INT(32) .EXT end_scan;  ! out
    INT     base;           ! in
                                TNS only
```

*str*

is a pointer to the string to convert.

*end\_scan*

is the address within *str* where the conversion ended. The pointer returned through *end\_scan* points to the first unrecognized character.

*base*

specifies the radix of the digits at *str*. It must be in the range:

$0 \leq \text{base} \leq 36$

- If *base* is zero, the leading characters of *str* determine the radix:
  - 0X or 0x indicates a hexadecimal number.
  - 0 indicates an octal number.
  - 1 through 9 indicate a decimal number.

- If *base* is greater than 1 and less than or equal to 36, it specifies the radix of the string of characters at *str*. For values of *base* greater than 10, the letter A represents 10, the letter B represents 11, and so forth, through the letter Z which represents 35. Strtol treats uppercase and lowercase alphabetic characters in *str* interchangeably.
- If *base* is less than 0, equal to 1, or greater than 36, the Strtol functions do not convert *str*. They return zero and *end\_scan* points to the same location as *str*.

## Return Value

The Strtol functions return the converted value.

## Considerations

- *str* must be terminated by a zero (null) byte to stop the scan.
- Leading white space is skipped.
- *str* is a string of digits, optionally preceded by a sign.
- Scanning stops if a character is encountered that is not valid for *base*.

## Example

```
STRING .s[0:7] := ["21B43",0];
STRING .end;
INT(32) d;

! d gets 138051, the decimal equivalent of hex 21B43.
! @end gets @s[5]

d := CRE_Strtol_(s, @end, 16);
```

## Strtol

The Strtol functions convert a string of characters to an unsigned 32-bit integer using a specified base.

```
INT(32) PROC CRE_Strtol_( str, end_scan, base );
    STRING      .str;          ! in
    INT          .end_scan;     ! out
    INT          base;          ! in                                TNS only

INT(32) PROC CRE_StrtolX_( str, end_scan, base );
    STRING .EXT str;          ! in
    INT(32) .EXT end_scan;    ! out
    INT          base;          ! in                                TNS only
```

*str*

points to the string to convert.

*end\_scan*

is the address within *str* where the conversion ended. The pointer returned through *end\_scan* points to the character immediately after the last converted character.

*base*

specifies the radix of the digits at *str*. It must be in the range:

0 less than or equal *base* less than or equal to 36

- If *base* is zero, the leading characters of *str* determine the radix:
  - 0X or 0x indicates a hexadecimal number.
  - 0 indicates an octal number.
  - 1 through 9 indicate a decimal number.
- If *base* is greater than 1 and less than or equal to 36, it specifies the radix of the string of characters at *str*. For values of *base* greater than 10, the letter A represents 10, the letter B represents 11, and so forth, through the letter Z which represents 35. Strtoul treats uppercase and lowercase alphabetic characters in *str* interchangeably.
- If *base* is less than 0, equal to 1, or greater than 36, the Strtoul functions do not convert *str* and return zero. *end\_scan* points to the same location as *str*.

## Return Value

The Strtoul functions return:

- The 32-bit converted value if the conversion is successful.
- Zero if they cannot convert the string.
- The highest value that can be represented in 32 bits if the converted value would cause overflow.

## Considerations

- Strtoul skips leading white-space characters.
- Strtoul stops scanning if it encounters a character that is not valid for *base*.

## Substring\_Search

The Substring\_Search function determines whether one string is a substring of another string. This function is not available in the native CRE library.

```
INT PROC RTL_Substring_Search_( substr, substr_len
                               str, str_len );
    STRING .EXT substr;      ! in
    INT     substr_len;      ! in
    STRING .EXT str;         ! in
    INT     str_len;         ! in
                                TNS only
```

*substr*

is a pointer to the substring to scan for in *str*.

*substr\_len*

is the length of *substr* in bytes.

*str*

is a pointer to the string to scan.

*str\_len*

is the length of *str* in bytes.

## Return Value

Substring\_Search returns the position within *str* at which *substr* begins, or zero if *substr* is not found within *str*. Note that the first character in *str* is at position one.

## Considerations

The ends of the strings *str* and *substr* are determined by *str\_len* and *substr\_len*, respectively. A zero (null) byte has no special meaning in this function.

## Example

```
INT count;
STRING .s[0:28] := ["HP Computers Incorporated"];
STRING .subs[0:8] := ["Computers"];

! count gets 8

count := RTL_Substring_Search_(subs, 9, s, 29);
```

# Memory Block Functions

This subsection describes the memory block functions. [Table 8-3](#) lists the memory block functions supported by the TNS CRE. Each function described in this section begins with the prefix `RTL_` or `CRE_`. Refer to [Using Standard Functions](#) on page 2-56 for more information.

**Table 8-3. TNS CRE Memory Block Functions**

Standard Function Name	Function Action
<code>memchr</code>	Searches for a character in a block of memory.
<code>memcmp</code>	Compares two blocks of memory.
<code>memcpy</code>	Copies a block of memory.
<code>memmove</code>	Moves a block of memory.
<code>memset</code>	Initializes a block of memory to a specified character value.
<code>memswap</code>	Exchanges one block of memory with a second block of memory.
<code>repmem</code>	Replicates values through a block of memory.

[Table 8-4](#) on page 8-33 shows the Memory Block functions supported by the native CRE library. The native CRE does not provide pTAL prototypes for these functions. It provides only the function names, which are case-sensitive. You can use the function prototypes in the C header files as examples when writing your own pTAL prototypes for these functions.

**Table 8-4. Native CRE Memory Block Functions**

Standard Function Name	Native CRE Library Function Action
<code>memchr</code>	Searches for a character in a block of memory.
<code>memcmp</code>	Compares two blocks of memory.
<code>memcpy</code>	Copies a block of memory.
<code>memmove</code>	Moves a block of memory.
<code>memset</code>	Initializes a block of memory to a specified character value.
<code>wmemchr</code>	Searches for a wide character in a block of memory.
<code>wmemcmp</code>	Compares two blocks of wide-character memory.
<code>wmemcpy</code>	Copies a block of wide-character memory.
<code>wmemmove</code>	Moves a block of wide-character memory.
<code>wmemset</code>	Initializes a block of wide-character memory to a specified character value.

## Memory\_Compare

The Memory\_Compare functions compare two blocks of memory. These functions are not available in the native CRE library.

```

INT PROC RTL_Memory_Compare_( buf1, buf2, num_bytes );
    STRING .buf1;           ! in
    STRING .buf2;           ! in
    INT     num_bytes;       ! in
                                TNS
only

INT PROC RTL_Memory_CompareX_( buf1, buf2, num_bytes );
    STRING .EXT buf1;       ! in
    STRING .EXT buf2;       ! in
    INT(32) num_bytes;      ! in
                                TNS
only

```

*buf1*

is a pointer to the first block of memory.

*buf2*

is a pointer to the second block of memory.

*num\_bytes*

specifies the number of bytes to compare.

## Return Value

Memory\_Compare returns a number that is:

```

< 0 if buf1 < buf2
  0 if buf1 = buf2
> 0 if buf1 > buf2

```

## Considerations

- The Memory\_Compare functions perform an unsigned byte comparison for a maximum count of *num\_bytes*.
- For less-than and greater-than comparisons, characters in *buf1* and *buf2* are compared using the characters' values in the ASCII collating sequence.

## Example

```

INT i;
STRING .a1[0:7] := "12345678";
STRING .a2[0:10] := "12344678";

i := RTL_Memory_Compare_(a1, a2, 8); ! i gets 1

```



## Memory\_Copy

The Memory\_Copy functions copy a block of memory. These functions are not available in the native CRE library.

```

INT PROC RTL_Memory_Copy_( dest, source, num_bytes );
    STRING .dest;                ! out
    STRING .source;              ! in
    INT     num_bytes;           ! in
only                                TNS

INT(32) PROC RTL_Memory_CopyX_( dest, source, num_bytes );
    STRING .EXT dest;            ! out
    STRING .EXT source;          ! in
    INT(32)    num_bytes;        ! in
only                                TNS

```

*dest*

is a pointer to the destination to which the block is copied.

*source*

is a pointer to the beginning of the block to copy.

*num\_bytes*

specifies the number of bytes to copy.

## Return Value

The Memory\_Copy functions return the address of the destination block, @*dest*.

## Considerations

The Memory\_Copy functions copy *source* to *dest* regardless of whether *source* and *dest* overlap. See also [Memory\\_Move](#) on page 8-37.

## Example

```

STRING .EXT dst := ( 4D '<<' 17 ) + 65536D;
STRING .EXT src := ( 4D '<<' 17 );

! copies first 64K bytes of extended segment

CALL RTL_Memory_CopyX_(dst, src, 65536D);

```

## Memory\_Findchar

The Memory\_Findchar functions search for a character in a block of memory. These functions are not available in the native CRE library.

```

INT PROC RTL_Memory_Findchar_( buf, char, num_bytes );
  STRING .buf;           ! in
  INT     char;           ! in
  INT     num_bytes;      ! in
                                TNS
only

INT(32) PROC RTL_Memory_FindcharX_( buf, char, num_bytes );
  STRING .EXT buf;       ! in
  INT     char;           ! in
  INT(32) num_bytes;      ! in
                                TNS
only

```

*buf*

is a pointer to the block of memory to search.

*char*

specifies the search character. Only bits <8:15> of *char* are used for the search character.

*num\_bytes*

specifies the number of bytes to search.

## Return Value

The Memory\_Findchar functions return a pointer to the location in *buf* where *char* was found. If *char* is not found, zero is returned.

## Example

```

STRING .buf[0:9] := [0,1,0,2,0,3,0,4,0,5];
INT     i;

i := RTL_Memory_Findchar_(buf, 2, 10); ! i gets @buf[3]
i := RTL_Memory_Findchar_(buf, 6, 10); ! i gets 0

```

## Memory\_Move

The Memory\_Move functions move a block of memory. These functions are not available in the native CRE library.

```

INT PROC RTL_Memory_Move_( dest, source, num_bytes );
    STRING .dest;                ! out
    STRING .source;              ! in
    INT     num_bytes;           ! in                                TNS
only

INT(32) PROC RTL_Memory_MoveX_( dest, source, num_bytes );
    STRING .EXT dest;            ! out
    STRING .EXT source;          ! in
    INT(32)   num_bytes;         ! in                                TNS
only

```

*dest*

is a pointer to the destination to which the block is moved.

*source*

is a pointer to the beginning of the block to move.

*num\_bytes*

specifies the number of bytes to move.

## Return Value

The Memory\_Move functions return the address of the destination block, @*dest*.

## Considerations

The Memory\_Move functions ensure that data is not lost, even if the source and destination blocks overlap.

## Example

```

STRING .EXT dst := ( 4D '<<' 17) + 65536D;
STRING .EXT src := ( 4D '<<' 17);

! move first 64k bytes of extended segment

CALL RTL_Memory_Move_(dst, src, 65536D);

```

## Memory\_Repeat

The Memory\_Repeat procedures fill a block of memory with a block of values a specified number of times. These functions are not available in the native CRE library. |

```
PROC RTL_Memory_Repeat_(dest, values, values_len, num_rep );
  STRING .dest;                ! out
  STRING .values;              ! in
  INT    values_len;          ! in
  INT    num_rep;              ! in                                TNS
only

PROC RTL_Memory_RepeatX_(dest, values, values_len, num_rep);
  STRING .EXT dest;            ! out
  STRING .EXT values;          ! in
  INT(32)  values_len;        ! in
  INT(32)  num_rep;           ! in                                TNS
only
```

*dest*

is a pointer to the block of memory to fill.

*values*

is a pointer to the block of values to replicate into *dest*.

*values\_len*

specifies the size of *values* in bytes.

*num\_rep*

specifies the number of times to replicate *values* into *dest*.

## Considerations

The Memory\_Repeat procedures are declared as procedures, not as functions. They do not return a value. They also ensure that data is not lost, even if the source and destination blocks overlap.

## Example

```
STRING .dst[0:499];
STRING .vals[0:9] := "repeat me ";
INT    len := 10;
INT    num := 50;
```

!vals is stored 50 times.

```
CALL RTL_Memory_Repeat_( dst, vals, len, num );
```

## Memory\_Set

The Memory\_Set functions initialize a block of memory to a specified character. These functions are not available in the native CRE library.

```

INT PROC RTL_Memory_Set_( buf, char, num_bytes );
    STRING .buf;          ! out
    INT     char;          ! in
    INT     num_bytes;     ! in
                                TNS
only

INT(32) PROC RTL_Memory_SetX_( buf, char, num_bytes );
    STRING .EXT buf;      ! out
    INT     char;          ! in
    INT(32) num_bytes;    ! in
                                TNS
only

```

*buf*

is a pointer to the block to initialize.

*char*

specifies the initialization character. Only bits <8:15> of *char* are used for the initialization value.

*num\_bytes*

specifies the number of bytes to initialize.

## Return Value

The Memory\_Set functions return the address of the block of memory, @*buf*.

## Considerations

- A concatenated move is faster than Memory\_Set.
- *char* must be a single character constant or a STRING variable.

## Example

```

STRING .EXT buf := 4D '<<' 17;
INT     ch      := 0;

! set first 64k bytes of extended segment to 0.

CALL RTL_Memory_SetX_(buf, ch, 65536D);

```

## Memory\_Swap

The Memory\_Swap functions exchange one block of memory with a second block of memory. These functions are not available in the native CRE library.

```
PROC RTL_Memory_Swap_( buf1, buf2, num_bytes );
  STRING .buf1;           ! in/out
  STRING .buf2;           ! in/out
  INT    num_bytes;       ! in
only                                     TNS

PROC RTL_Memory_SwapX_( buf1, buf2, num_bytes );
  STRING .EXT buf1;       ! in/out
  STRING .EXT buf2;       ! in/out
  INT(32) num_bytes;      ! in
only                                     TNS
```

*buf1*

is a pointer to the first block of memory.

*buf2*

is a pointer to the second block of memory.

*num\_bytes*

specifies the number of bytes to exchange.

## Considerations

The Memory\_Swap procedures are declared as procedures, not as functions. They do not return a value.

## Example

```
STRING .up[0:9]    := "0123456789";
STRING .down[0:9]  := "9876543210";

CALL RTL_Memory_Swap_( up, down, 10 );
```

# Common Language Utility (CLU) Library Functions

This section presents a set of Common Language Utility (CLU) library functions provided in the CRE. This section describes functions that enable:

- COBOL and FORTRAN routines to create processes
- COBOL and FORTRAN routines to locate and identify file connectors
- COBOL, FORTRAN, TAL, and pTAL routines to save and manipulate messages sent to a process by the process that initiated the process

CLU functions are separate and different from CRE functions:

- TNS CRE functions are located in the file CRELIB. TNS CLU functions are located in the file CLULIB.
- TNS CRE functions are configured into the system library. TNS CLU functions must be explicitly bound into programs that use them.
- CRE functions work only in the CRE. CLU functions work in the CRE or in certain language-specific run-time environments.

Two CLU routines are described in this section: [CLU\\_Process\\_Create](#) on page 9-1 and [CLU\\_Process\\_File\\_Name](#) on page 9-12.

The CLU routines are available only to processes running in the Guardian environment. Processes running in the OSS environment can use the CRE\_Getenv\_ and CRE\_Putenv\_ functions, described in [Section 6, CRE Service Functions](#), to obtain environment information.

The SMU functions are listed in [Table 9-1](#) on page 9-18 and are described on the pages following the table.

## CLU\_Process\_Create\_

The CLU\_Process\_Create\_ function provides a facility that creates a process using the conventions of the TACL RUN command.

Both the TNS CRE and the native CRE support this function. You can use this function either in the CRE or in a COBOL or FORTRAN run-time environment.

CLU\_Process\_Create\_ first calls the system procedure PROCESS\_CREATE\_, passing through all specified parameters. If process creation is successful, CLU\_Process\_Create\_ sends appropriate process initialization messages, depending upon the options specified in the parameter *clu\_options*. Most PROCESS\_CREATE\_ functionality is available; however, CLU\_Process\_Create\_ rejects a request for no-wait process creation.

The function declaration is presented first, followed by COBOL and FORTRAN considerations. Here is the function declaration:

```

INT PROC CLU_Process_Create_( clu_options,
                             program_file:program_file_bytes,
                             library_file:library_file_bytes,
                             swap_file:swap_file_bytes,
                             ext_swap_file:ext_swap_file_bytes,
                             priority, processor, processhandle,
                             error_detail, name_option,
                             name:name_bytes,
                             process_descr:process_descr_maxbytes,
                             nowait_tag, hometerm:hometerm_bytes,
                             memory_pages, jobid, create_options,
                             defines:defines_bytes, debug_options,
                             pfs_size ) EXTENSIBLE;

INT      clu_options;           !in, optional
STRING .EXT program_file;      !in, required
INT      program_file_bytes;    !in, required
STRING .EXT library_file;      !in, optional
INT      library_file_bytes;    !in, optional
STRING .EXT swap_file;         !in, optional
INT      swap_file_bytes;       !in, optional
STRING .EXT ext_swap_file;     !in, optional
INT      ext_swap_file_bytes;   !in, optional
INT      priority;              !in, optional
INT      processor;             !in, optional
INT      .EXT processhandle;    !out, optional
INT      .EXT error_detail;     !out, optional
INT      name_option;           !in, optional
STRING .EXT name;              !in, optional
INT      name_bytes;            !in, optional
STRING .EXT process_descr;     !out, optional
INT      process_descr_maxbytes; !in, optional
INT      .EXT process_descr_bytes; !out, optional
INT(32)  nowait_tag;            !in, optional
STRING .EXT hometerm;          !in, optional
INT      hometerm_bytes;        !in, optional
INT      memory_pages;         !in, optional
INT      jobid;                 !in, optional
INT      create_options;        !in, optional
STRING .EXT defines;           !in, optional
INT      defines_bytes;         !in, optional
INT      debug_options;         !in, optional
INT(32)  pfs_size;              !in, optional TNS, native

```



*clu\_options*

if present, specifies function options. The bits in *clu\_options* are defined as follows:

Bits in <i>clu_options</i>	Meaning
12	PROCESS_STOP option:  0: If PROCESS_CREATE_ returns 14 (undefined externals), stop the created process and return 14 as the function value.  1: If PROCESS_CREATE_ returns 14 (undefined externals), consider process creation successful. Unless a failure occurs in a later step, return 14 as the function value.
13:15	Send messages option:  0: Send copies of all process initialization messages saved by the creator process (default).  1: Send a copy of the saved Startup message only—do not send any ASSIGN or PARAM messages.  2: Send a default Startup message. The saved Startup message is ignored.  3: Send no process initialization message. The creator process does not open the created process because it has nothing to communicate.

For *clu\_options* .<12>, if no saved Startup message exists, create one using the creator process attributes for default volume/subvolume, IN name, and OUT name, and supply a null string for the message parameter string value.

*program\_file*

specifies the name of the new process program file.

*program\_file\_bytes*

is the size, in bytes, of *program\_file* . For COBOL and FORTRAN, the compiler implicitly computes and transmits this value to the function.

All subsequent parameters are passed through to the system procedure PROCESS\_CREATE\_ as the parameters to which they correspond. See the PROCESS\_CREATE\_ procedure in the *Guardian Procedure Calls Reference Manual* for explanations of the purpose and effect of these parameters.

CLU\_Process\_Create\_ scans each string input parameter; if it finds a blank (space character), it truncates the string at that point—that is, it discards the blank and all following characters.

# Return Value

CLU\_Process\_Create\_ returns one of the following values:

Return Value	Meaning
-4	A file management error occurred due to failure of FILE_OPEN or WRITEREAD [X]; the error value is returned in <i>error_detail</i> .
-3	A Startup message cannot be made because the IN file, OUT file, or default volume name cannot be edited into the corresponding message field (this can occur if the name cannot be converted to network form).
-2	An invalid parameter value.
-1	A required parameter is missing, or some internal logic error occurred.
0	The operation is successful.
> 0	An error code from PROCESS_CREATE_; the error value is returned in <i>error_detail</i> .

# COBOL Considerations

You can use CLU\_Process\_Create\_ for programs that run in the CRE or in a COBOL run-time environment. The COBOL format for invoking this function is:

```
ENTER [ TAL ] "CLU_Process_Create_"
  [ OF library-reference ]
  USING clu-options, program-file, library-file, swap-file,
  ext-swap-file, priority, processor, processhandle,
  error-detail, name-option, name, process-descr,
  process-descr-bytes, nowait-tag, hometerm, memory-pages,
  jobid, create-options, defines, debug-options, pfs-size
  [ GIVING result ]
```

TAL

Generates more efficient code.

*library-reference*

is as described in the *COBOL Manual for TNS and TNS/R Programs*, except that it is contained in the file CLULIB (which is usually in the \$SYSTEM.SYSTEM subvolume), rather than in a COBOL85 product file such as COBOLLIB.

*clu-options*

is an actual parameter that evaluates to a numeric value appropriate for the PROCESS\_CREATE\_ parameter to which it corresponds, such as a COMPUTATIONAL, DISPLAY, or NATIVE numeric data item, a numeric literal, or an expression enclosed in parentheses. If the parameter does not mathematically evaluate to an integer, the compiler changes the result into an integer value.

*program-file*

is an alphanumeric operand containing a string value appropriate for the `PROCESS_CREATE_` parameter to which it corresponds.

The compiler implicitly computes and transmits the size, in bytes, of *program-file* to the function.

If a string value does not completely fill its operand, the first unused character must be a blank (space character).

*library-file*

is an alphanumeric operand containing a string value appropriate for the `PROCESS_CREATE_` parameter to which it corresponds.

The compiler implicitly computes and transmits the size, in bytes, of *library-file* to the function.

If a string value does not completely fill its operand, the first unused character must be a blank (space character).

*swap-file*

is an alphanumeric operand containing a string value appropriate for the `PROCESS_CREATE_` parameter to which it corresponds.

The compiler implicitly computes and transmits the size, in bytes, of *swap-file* to the function.

If a string value does not completely fill its operand, the first unused character must be a blank (space character).

*ext-swap-file*

is an alphanumeric operand containing a string value appropriate for the `PROCESS_CREATE_` parameter to which it corresponds.

The compiler implicitly computes and transmits the size, in bytes, of *ext-swap-file* to the function.

If a string value does not completely fill its operand, the first unused character must be a blank (space character).

*priority*

is an actual parameter that evaluates to a numeric<sup>1</sup> value appropriate for the `PROCESS_CREATE_` parameter to which it corresponds, such as a `COMPUTATIONAL`, `DISPLAY`, or `NATIVE` numeric data item, a numeric literal, or an expression enclosed in parentheses. If the parameter does not mathematically evaluate to an integer, the compiler changes the result into an integer value.

*processor*

is an actual parameter that evaluates to a numeric<sup>1</sup> value appropriate for the `PROCESS_CREATE_` parameter to which it corresponds, such as a `COMPUTATIONAL`, `DISPLAY`, or `NATIVE` numeric data item, a numeric literal, or an expression enclosed in parentheses. If the parameter does not mathematically evaluate to an integer, the compiler changes the result into an integer value.

*processhandle*

is a table with at least ten occurrences, where an occurrence is a `NATIVE-2` data item or a `COMPUTATIONAL` numeric data item described with one to four 9s preceded by an S; for example:

```
01 processhandle occurs 10 times.
   05 filler native-2.
```

*error-detail*

is a `NATIVE-2` data item or a `COMPUTATIONAL` numeric data item described with five to nine 9s preceded by an S; for example:

```
PICTURE S9(4)
```

*name-option*

is an actual parameter that evaluates to a numeric<sup>1</sup> value appropriate for the `PROCESS_CREATE_` parameter to which it corresponds, such as a `COMPUTATIONAL`, `DISPLAY`, or `NATIVE` numeric data item, a numeric literal, or an expression enclosed in parentheses. If the parameter does not mathematically evaluate to an integer, the compiler changes the result into an integer value.

*name*

is an alphanumeric operand containing a string value appropriate for the `PROCESS_CREATE_` parameter to which it corresponds.

The compiler implicitly computes and transmits the size, in bytes, of *name* to the function.

If a string value does not completely fill its operand, the first unused character must be a blank (space character).

*process-descr*

is an alphanumeric data item.

The compiler implicitly computes and transmits the size, in bytes, of *process-descr* to the function.

If the size of the assigned value differs from the size of the data item, the text is truncated or extended with blanks (space characters) as necessary to fill the data item. If the data item is described with a `JUSTIFIED` clause, that clause is ignored.

*process-descr-bytes*

is a NATIVE-2 data item or a COMPUTATIONAL numeric data item described with one to four 9s preceded by an S; for example:

```
PICTURE S9(4)
```

*nowait-tag*

is an actual parameter that evaluates to a numeric1 value appropriate for the PROCESS\_CREATE\_ parameter to which it corresponds, such as a COMPUTATIONAL, DISPLAY, or NATIVE numeric data item, a numeric literal, or an expression enclosed in parentheses. If the parameter does not mathematically evaluate to an integer, the compiler changes the result into an integer value.

*hometerm*

is an alphanumeric operand containing a string value appropriate for the PROCESS\_CREATE\_ parameter to which it corresponds.

The compiler implicitly computes and transmits the size, in bytes, of *hometerm* to the function.

If a string value does not completely fill its operand, the first unused character must be a blank (space character).

*memory-pages*

is an actual parameter that evaluates to a numeric1 value appropriate for the PROCESS\_CREATE\_ parameter to which it corresponds, such as a COMPUTATIONAL, DISPLAY, or NATIVE numeric data item, a numeric literal, or an expression enclosed in parentheses. If the parameter does not mathematically evaluate to an integer, the compiler changes the result into an integer value.

*jobid*

is an actual parameter that evaluates to a numeric1 value appropriate for the PROCESS\_CREATE\_ parameter to which it corresponds, such as a COMPUTATIONAL, DISPLAY, or NATIVE numeric data item, a numeric literal, or an expression enclosed in parentheses. If the parameter does not mathematically evaluate to an integer, the compiler changes the result into an integer value.

*create-options*

is an actual parameter that evaluates to a numeric1 value appropriate for the PROCESS\_CREATE\_ parameter to which it corresponds, such as a COMPUTATIONAL, DISPLAY, or NATIVE numeric data item, a numeric literal, or an expression enclosed in parentheses. If the parameter does not mathematically evaluate to an integer, the compiler changes the result into an integer value.

*defines*

is an alphanumeric operand containing a string value appropriate for the `PROCESS_CREATE_` parameter to which it corresponds.

The compiler implicitly computes and transmits the size, in bytes, of *defines* to the function.

If a string value does not completely fill its operand, the first unused character must be a blank (space character).

*debug-options*

is an actual parameter that evaluates to a numeric value appropriate for the `PROCESS_CREATE_` parameter to which it corresponds, such as a `COMPUTATIONAL`, `DISPLAY`, or `NATIVE` numeric data item, a numeric literal, or an expression enclosed in parentheses. If the parameter does not mathematically evaluate to an integer, the compiler changes the result into an integer value.

*pfs-size*

is an actual parameter that evaluates to a numeric value appropriate for the `PROCESS_CREATE_` parameter to which it corresponds, such as a `COMPUTATIONAL`, `DISPLAY`, or `NATIVE` numeric data item, a numeric literal, or an expression enclosed in parentheses. If the parameter does not mathematically evaluate to an integer, the compiler changes the result into an integer value.

*result*

is a numeric data item.

## FORTRAN Considerations

You can use `CLU_Process_Create_` for programs that run in the TNS CRE or in a FORTRAN run-time environment.

Before you use this function, you must declare it in a `CONSULT` directive that specifies an object file that contains a copy of the function. This function is contained in the file `CLULIB` (which is usually in the `$SYSTEM.SYSTEM` subvolume), rather than in a FORTRAN product file.

The FORTRAN format for invoking this function is as follows:

```
result = CLU_Process_Create_ ( cluoptions, programfile,
                               libraryfile, swapfile, extswapfile, priority,
                               processor, processhandle, errordetail,
                               nameoption, name, processdescr,
                               processdescrbytes, nowaittag, hometerm,
                               memorypages, jobid, createoptions, defines,
                               debugoptions, pfssize )
```

*result*

is a 2-byte integer variable (INTEGER\*2).

*cluoptions*

is an integer expression that evaluates to a numeric value appropriate for the PROCESS\_CREATE\_ parameter to which it corresponds.

*programfile*

is a character operand containing a string value appropriate for the PROCESS\_CREATE\_ parameter to which it corresponds.

The compiler implicitly computes and transmits the size, in bytes, of *programfile* to the function.

If a string value does not completely fill its operand, the first unused character must be a blank (space character).

*libraryfile*

is a character operand containing a string value appropriate for the PROCESS\_CREATE\_ parameter to which it corresponds.

The compiler implicitly computes and transmits the size, in bytes, of *libraryfile* to the function.

If a string value does not completely fill its operand, the first unused character must be a blank (space character).

*swapfile*

is a character operand containing a string value appropriate for the PROCESS\_CREATE\_ parameter to which it corresponds.

The compiler implicitly computes and transmits the size, in bytes, of *swapfile* to the function.

If a string value does not completely fill its operand, the first unused character must be a blank (space character).

*extswapfile*

is a character operand containing a string value appropriate for the PROCESS\_CREATE\_ parameter to which it corresponds.

The compiler implicitly computes and transmits the size, in bytes, of *extswapfile* to the function.

If a string value does not completely fill its operand, the first unused character must be a blank (space character).

*priority*

is an integer expression that evaluates to a numeric value appropriate for the `PROCESS_CREATE_` parameter to which it corresponds.

*processor*

is an integer expression that evaluates to a numeric value appropriate for the `PROCESS_CREATE_` parameter to which it corresponds.

*processhandle*

is a 2-byte integer array with at least ten elements; for example:

`INTEGER*2 PROCESSHANDLE (10).`

*errordetail*

is a 2-byte integer variable (`INTEGER*2`).

*nameoption*

is an integer expression that evaluates to a numeric value appropriate for the `PROCESS_CREATE_` parameter to which it corresponds.

*name*

is a character operand containing a string value appropriate for the `PROCESS_CREATE_` parameter to which it corresponds.

The compiler implicitly computes and transmits the size, in bytes, of *name* to the function.

If a string value does not completely fill its operand, the first unused character must be a blank (space character).

*processdescr*

is a `CHARACTER` variable.

The compiler implicitly computes and transmits the size, in bytes, of *processdescr* to the function.

When the size of the assigned value differs from the size of the variable, the text is truncated or extended with blanks as necessary to fill the variable.

*processdescrbytes*

is a 2-byte integer variable (`INTEGER*2`).

*nowaittag*

is an integer expression that evaluates to a numeric value appropriate for the `PROCESS_CREATE_` parameter to which it corresponds.



*hometerm*

is a character operand containing a string value appropriate for the PROCESS\_CREATE\_ parameter to which it corresponds.

The compiler implicitly computes and transmits the size, in bytes, of *hometerm* to the function.

If a string value does not completely fill its operand, the first unused character must be a blank (space character).

*memorypages*

is an integer expression that evaluates to a numeric value appropriate for the PROCESS\_CREATE\_ parameter to which it corresponds.

*jobid*

is an integer expression that evaluates to a numeric value appropriate for the PROCESS\_CREATE\_ parameter to which it corresponds.

*createoptions*

is an integer expression that evaluates to a numeric value appropriate for the PROCESS\_CREATE\_ parameter to which it corresponds.

*defines*

is a character operand containing a string value appropriate for the PROCESS\_CREATE\_ parameter to which it corresponds.

The compiler implicitly computes and transmits the size, in bytes, of *defines* to the function.

If a string value does not completely fill its operand, the first unused character must be a blank (space character).

*debugoptions*

is an integer expression that evaluates to a numeric value appropriate for the PROCESS\_CREATE\_ parameter to which it corresponds.

*pfssize*

is an integer expression that evaluates to a numeric value appropriate for the PROCESS\_CREATE\_ parameter to which it corresponds.

# CLU\_Process\_File\_Name\_

The CLU\_Process\_File\_Name\_ function processes COBOL and FORTRAN file connectors. The native CRE does not support this function. You can use this function either in the TNS CRE or in a COBOL or FORTRAN run-time environment.

The function declaration is presented first, followed by COBOL and FORTRAN considerations. Here is the function declaration:

```
INT PROC CLU_Process_File_Name_( call_type, fcb_address,
                                tdm_file_name:tdm_file_name_size,
                                file_name:file_name_size, open_flag )
extensible;
  INT      call_type;           ! in,      required
  INT(32)  .EXT fcb_address;     ! in/out, required
  STRING   .EXT tdm_file_name;  ! in/out, required
  INT      tdm_file_name_size; ! in,      required
  STRING   .EXT file_name;      ! out,     optional
  INT      file_name_size;     ! in,      optional
  INT      .EXT open_flag;      ! out,     optional
                                           ! TNS only
```

## call\_type

identifies the purpose of the call. *call\_type* must be either:

- 0   Retrieves current attribute information.
- 1   Specifies new attribute information; the specified File Control Block (FCB) must be closed and a valid backup process must not exist.

## fcb\_address

is the address of an FCB. The input value must be either:

Input Value	call_type	Meaning
0	0	Retrieves current information about first FCB in run unit
FCB address obtained from a previous call to this function	0	Retrieves current information about next FCB in run-unit
FCB address obtained from a previous call to this function	1	Specifies information about next FCB in run-unit

The output value is the address of the located FCB. The ordering of FCBs for the TNS CRE, however, differs from the ordering for a COBOL or FORTRAN run-time environment; the first or next FCB might differ depending on the environment. If no first or next FCB exists, the output is -1.

*tdm\_file\_name*

- If *call\_type* is 0, *tdm\_file\_name* is a variable to contain the retrieved Guardian file name of the located FCB.

The retrieved file name is in external format; for example, \$A.B.C or =XYZ. If the variable is larger than the file name, the function left-justifies the name and fills the remainder of the variable with space characters. If the variable is smaller than the name, the function truncates the name on the right and returns a value of -1. (Any COBOL JUSTIFIED clause in the variable declaration is ignored.) If the current name is a DEFINE name, the function returns its text, not the associated disk file name or device name, if any.

- If *call\_type* is 1, *tdm\_file\_name* is a new Guardian file name to assign to the located FCB.

Specify a valid Guardian file name in external format; for example, \$A.B.C or =XYZ. If the variable is larger than the name, you must fill the remainder of the variable with space characters. The function ignores trailing space characters.

*tdm\_file\_name\_size*

is the size, in bytes, of *tdm\_file\_name*. For TAL, you must specify this value. For COBOL and FORTRAN, the compiler implicitly computes and transmits this value to the function.

*file\_name*

- If *call\_type* is 0, *file\_name* is a variable to contain the retrieved logical file name of the located FCB.
- If *call\_type* is 1, *file\_name* is ignored.

*file\_name\_size*

is the size, in bytes, of *file\_name*. For TAL, you must specify this value if *file\_name* is present. For COBOL and FORTRAN, the compiler implicitly computes and transmits this value to the function.

*open\_flag*

If *call\_type* is 0, *open\_flag* is a variable to contain the retrieved open mode value of the FCB. The retrieved value is one of the following:

- 0   Closed    The FCB is not in an open mode.
- 1   Locked    The FCB is not in an open mode.  
                  A COBOL variant of Closed.
- 2   Input     The FCB is open for input operations.
- 4   Output    The FCB is open for output operations.
- 5   Extend    The FCB is open for output operations.  
                  A COBOL variant of Output.
- 6   I-O       The FCB is open for input and output operations.

If *call\_type* is 1, *open\_flag* is ignored.

Return Value

CLU\_Process\_File\_Name\_ returns one of the following values:

Return Value	Meaning
-2	The operation is successful, but the retrieved logical file name is truncated.
-1	The operation is successful, but the retrieved Guardian file name is truncated.
0	The operation is successful.
1	A required parameter is missing.
2	A parameter value is invalid. For example: <ul style="list-style-type: none"><li>● <i>call_type</i> is neither 0 nor 1.</li><li>● <i>fcg_address</i> is neither 0 nor the address of an FCB.</li><li>● <i>tdm_file_name_size</i> or <i>file_name_size</i> is less than 0.</li></ul>
3	The Guardian file name is invalid.
4	The run-time environment is invalid (not TNS CRE or COBOL/FORTRAN run-time environment) or has been corrupted.
5	<i>call_type</i> is 1 and the FCB is open.
6	<i>call_type</i> is 1 and a valid backup process exists.

## COBOL Considerations

You can use `CLU_Process_File_Name_` for programs that run in the TNS CRE or in a COBOL run-time environment. The COBOL format for invoking this function is:

```
ENTER [ TAL ] "CLU_Process_File_Name_"
      [ OF library-reference ]
      USING call-type, fcf-address, tdm-file-name,
           file-name, open-flag
      [ GIVING result ]
```

*TAL*

Generates more efficient code.

*library-reference*

is as described in the *COBOL Manual for TNS and TNS/R Programs*, except that it is contained in the file `CLULIB` (which is usually in the `$SYSTEM.SYSTEM` subvolume), rather than in a COBOL85 product file such as `COBOLLIB`.

*call-type*

is an actual parameter that evaluates to 0 or 1, such as a COMPUTATIONAL, DISPLAY, or NATIVE numeric data item, a numeric literal, or an expression enclosed in parentheses. If the parameter does not mathematically evaluate to an integer, the compiler changes the result into an integer value. To specify conversion options, you can use COBOL functions `INTEGER` and `INTEGER_PART` (D10 release or later).

*fcf-address*

is a NATIVE-4 data item or a COMPUTATIONAL numeric data item described with five to nine 9s preceded by an S; for example:

```
PICTURE S9(9)
```

*tdm-file-name*

If *call-type* is 0, *tdm-file-name* is an alphanumeric data item to contain the retrieved Guardian file name.

If *call-type* is 1, *tdm-file-name* is an alphanumeric value that supplies a Guardian file name.

The compiler implicitly computes and transmits the size, in bytes, of *tdm-file-name* to the function.

*file-name*

If *call-type* is 0, *file-name* is an alphanumeric data item to contain the retrieved logical file name.

If *call-type* is 1, *file-name* is ignored.

The compiler implicitly computes and transmits the size, in bytes, of *file-name* to the function.

*open-flag*

is a NATIVE-2 data item or a COMPUTATIONAL numeric data item described with one to four 9s preceded by an S; for example:

```
PICTURE S9(9)
```

*result*

is a numeric data item.

A file connector has a logical file name and a Guardian file name:

- The compiler obtains the logical file name of a file connector from the file name specified in the SELECT clause of a file control entry and repeated after the FD in the associated file description entry.
- The compiler assigns the default Guardian file name of a file connector as described in the *COBOL Manual for TNS and TNS/R Programs*.

## FORTRAN Considerations

You can use `CLU_Process_File_Name_` for programs that run in the TNS CRE or in a FORTRAN run-time environment.

Before you use this function, you must declare it in a CONSULT directive that specifies an object file that contains a copy of the function. This function is contained in the file `CLULIB` (which is usually in the `$SYSTEM.SYSTEM` subvolume), rather than in a FORTRAN product file.

The FORTRAN format for invoking this function is as follows:

```
result = CLU_Process_File_Name_ ( calltype, fcbaddress,
                                tdmfilename, filename, openflag )
```

*result*

is a 2-byte integer variable (INTEGER\*2).

*calltype*

is an integer expression that evaluates to 0 or 1.

*fcaddress*

is a 4-byte integer variable (INTEGER\*4).

*tdmfilename*

If *calltype* is 0, *tdmfilename* is a character variable to contain the retrieved Guardian file name.

If *calltype* is 1, *tdmfilename* is a character value that supplies a Guardian file name.

The compiler implicitly computes and transmits the size, in bytes, of *tdmfilename* to the function.

*filename*

If *calltype* is 0, *filename* is a character variable to contain the retrieved logical file name.

If *calltype* is 1, *filename* is ignored.

The compiler implicitly computes and transmits the size, in bytes, of *filename* to the function.

*openflag*

is a 2-byte integer variable (INTEGER\*2).

A file connector has a logical file name and a Guardian file name:

- The compiler obtains the logical file name of a file connector from the UNIT directive describing unit *nnn* (if specified). Otherwise, the compiler constructs the logical file name FT*nnn*; for example, FT010 is the default logical file name text for unit 10.
- The compiler assigns the default Guardian file name of a file connector as described in the *FORTRAN Reference Manual*.

## SMU Functions

The SMU functions enable COBOL, FORTRAN, and TAL routines to manipulate saved ASSIGN, PARAM, and startup messages that are sent to your process when it is initiated.

All SMU functions are available in the native CRE library. The SMU functions are described in the following pages, the last of which is followed by COBOL, FORTRAN, and TAL considerations that apply to all the SMU functions.

**Table 9-1. SMU Functions**

<b>Name</b>	<b>Action</b>
<a href="#">SMU_Assign_CheckName_</a> on page 9-18	Checks whether an ASSIGN message with a given logical file name exists.
<a href="#">SMU_Assign_Delete</a> on page 9-19	Deletes a portion or all of an ASSIGN message.
<a href="#">SMU_Assign_GetText</a> on page 9-21	Retrieves a portion of an ASSIGN message as text and assigns it to a string variable.
<a href="#">SMU_Assign_GetValue</a> on page 9-22	Retrieves a portion of an ASSIGN message as an integer and assigns it to an integer variable.
<a href="#">SMU_Assign_PutText</a> on page 9-23	Creates or replaces a portion of an ASSIGN message with text from a string variable.
<a href="#">SMU_Assign_PutValue</a> on page 9-25	Creates or replaces a portion of an ASSIGN message with a value from an integer variable.
<a href="#">SMU_Message_CheckNumber_</a> on page 9-26	Checks whether a specific message exists.
<a href="#">SMU_Param_Delete</a> on page 9-27	Deletes a portion or all of the PARAM message.
<a href="#">SMU_Param_GetText</a> on page 9-28	Retrieves a portion of the PARAM message as text and assigns it to a string variable.
<a href="#">SMU_Param_PutText</a> on page 9-29	Creates or replaces a portion of a PARAM message with text from a string variable.
<a href="#">SMU_Startup_Delete</a> on page 9-30	Deletes the entire startup message.
<a href="#">SMU_Startup_GetText</a> on page 9-31	Retrieves a portion of the startup message as text and assigns it to a string variable.
<a href="#">SMU_Startup_PutText</a> on page 9-33	Creates or replaces a portion of the startup message with text from a string variable.

## SMU\_Assign\_CheckName\_

The `SMU_Assign_CheckName_` function checks whether a saved ASSIGN message with a given logical file name exists. It also returns the message number of the saved ASSIGN message.

```

INT PROC SMU_Assign_CheckName_ ( name:name_bytes )
                                EXTENSIBLE;
    STRING .EXT name;          ! in,    required
    INT       name_bytes;      ! in,    required TNS,native

```



*name*

is a logical file name of an ASSIGN message. If *name* includes a program name, *name* must have one of the following forms (maximum 63 characters):

*programname.filename*

*\*.filename*

If *name* does not include a program name, *name* must have the following form (maximum 31 characters):

*filename*

*name\_bytes*

is the size of *name*, in bytes, and must have a value greater than 0. For TAL, you must supply this value; for COBOL and FORTRAN, the compiler implicitly computes and transmits this value to the function. If *name\_bytes* exceeds the number of characters in *name*, the first unused character of *name* must be a space character.

Return Value

SMU\_Assign\_CheckName\_ returns one of the following values:

Return Value	Meaning
< 0	The negated message number of the first ASSIGN message whose logical file name conflicts with the one supplied; that is, one is qualified and the other is not, or one is qualified by * and the other is qualified by a program name.
0	The operation failed because of a logic error, such as: <ul style="list-style-type: none"><li>● No ASSIGN message with the specified logical file name exists.</li><li>● <i>name_bytes</i> is not greater than 0.</li><li>● <i>name</i> does not follow the required format.</li></ul>
> 0	The operation is successful; the return value gives the message number of the located ASSIGN message.

SMU\_Assign\_Delete\_

The SMU\_Assign\_Delete\_ function deletes a part or all of an ASSIGN message.

```
INT PROC SMU_Assign_Delete_ (message_number,
                             portion:portion_bytes )
                             EXTENSIBLE;
    INT      message_number;      ! in,  required
    STRING .EXT portion;          ! in,  required
    INT      portion_bytes;       ! in,  required TNS,native
```

*message\_number*

is an integer expression that identifies an ASSIGN message. *message\_number* must be a value greater than 0.

*portion*

is the identifier of an ASSIGN message integer part to delete. *portion* is one of:

ACCESS	Access mode—0 for read-write, 1 for read-only, 2 for write-only, and 3 for extend
BLKSIZE	Block mode size
EXCLUSION	Exclusion mode—0 for shared, 1 for protected, and 3 for exclusive
FILECODE	File code
PRIEXT	Primary extent size
RECSIZE	Record size
SECEXT	Secondary extent size
TANDEMNAME	Guardian file name
*ALL*	The entire ASSIGN message

*portion\_bytes*

is the size of *portion*, in bytes, and must be a value greater than 0. For TAL, you must supply this value; for COBOL and FORTRAN, the compiler implicitly computes and transmits this value to the function. If *portion\_bytes* exceeds the number of characters in *portion*, the first unused character of *portion* must be a space character.

## Return Value

SMU\_Assign\_Delete\_ returns one of the following values:

Return Value	Meaning
0	The operation is successful.
-1	The operation failed because of a logic error, such as: <ul style="list-style-type: none"><li>● <i>message_number</i> or <i>portion_bytes</i> has an invalid value.</li><li>● The specified ASSIGN message does not exist.</li><li>● <i>portion</i> does not identify a defined deletable part of an ASSIGN message.</li></ul>
-2	The operation failed because making changes would invalidate the backup process.

## SMU\_Assign\_GetText\_

The `SMU_Assign_GetText_` function retrieves a part of an ASSIGN message as text and assigns it to a string variable.

```

INT PROC SMU_Assign_GetText_( message_number,
                             portion:portion_bytes,
                             text:max_text_bytes )
                             EXTENSIBLE;
    INT      message_number;      ! in,  required
    STRING   .EXT portion;        ! in,  required
    INT      portion_bytes;       ! in,  required
    STRING   .EXT text;          ! out, optional
    INT      max_text_bytes;      ! in,  optional TNS,native

```

*message\_number*

is an integer expression that identifies the ASSIGN message from which to retrieve text. *message\_number* must have a value greater than 0.

*portion*

identifies what you want to retrieve. *portion* is one of:

**LOGICALNAME**      Retrieves the logical file name. If it includes a program name, the text has the following form (maximum 63 characters):

*programname.filename*

If it omits a program name, the text has the following form (maximum 31 characters):

*filename*

**TANDEMNAME**      Retrieves the Guardian file name (maximum 34 characters), which can be all blanks.

**\*ALL\***              Retrieves the entire ASSIGN message as a string (a sequence of 108 characters).

*portion\_bytes*

is the size of *portion*, in bytes, and must have a value greater than 0. For TAL, you must supply this value; for COBOL and FORTRAN, the compiler implicitly computes and transmits this value to the function. If *portion\_bytes* exceeds the number of characters in *portion*, the first unused character of *portion* must be a space character.

*text*

is a variable to contain the retrieved text. The function extends *text* with blanks on the right side or truncates it as necessary to match the size of the variable.

*max\_text\_bytes*

is the maximum size of *text*, in bytes, and must be a non-negative value. For TAL, you must supply this value if *text* is present; for COBOL and FORTRAN, the compiler implicitly computes and transmits this value to the function.

Return Value

SMU\_Assign\_GetText\_ returns one of the following values:

Return Value	Meaning
0	The operation is successful. The retrieved Tandem file name is all blanks.
> 0	The operation is successful. The return value gives the length, in bytes, of the retrieved text before truncation or padding.
-1	The operation failed because of a logic error, such as: <ul style="list-style-type: none"><li>• <i>message_number</i>, <i>portion_bytes</i>, or <i>max_text_bytes</i> has an invalid value.</li><li>• The specified ASSIGN message does not exist.</li><li>• <i>portion</i> does not identify a defined textual part of an ASSIGN message or identifies a nonexistent part.</li></ul>

SMU\_Assign\_GetValue\_

The SMU\_Assign\_GetValue\_ function retrieves a part of an ASSIGN message as an integer and assigns it to a numeric variable.

```
INT PROC SMU_Assign_GetValue_ ( message_number,
                                portion:portion_bytes, value )
                                EXTENSIBLE;
    INT          message_number;          ! in,  required
    STRING       .EXT portion;             ! in,  required
    INT          portion_bytes;            ! in,  required
    INT          .EXT value;                ! out, optional TNS,native
```

*message\_number*

is an integer expression that identifies the ASSIGN message from which to retrieve a value. *message\_number* must have a value greater than 0.

*portion*

is the identifier of an ASSIGN message integer part to retrieve. *portion* is one of:

- ACCESS        Access mode —0 for read-write, 1 for read-only, 2 for write-only, and 3 for extend
- BLKSIZE       Block mode size
- EXCLUSION     Exclusion mode —0 for shared, 1 for protected, and 3 for exclusive

FILECODE	File code
PRIEXT	Primary extent size
RECSIZE	Record size
SECEXT	Secondary extent size

*portion\_bytes*

is the size of *portion*, in bytes, and must be a value greater than 0. For TAL, you must supply this value; for COBOL and FORTRAN, the compiler implicitly computes and transmits this value to the function. If *portion\_bytes* exceeds the number of characters in *portion*, the first unused character of *portion* must be a space character.

*value*

is a variable to contain the retrieved value.

## Return Value

Values returned by SMU\_Assign\_GetValue \_ are:

Return Value	Meaning
0	The operation is successful.
-1	The operation failed because of a logic error, such as: <ul style="list-style-type: none"><li>• <i>message_number</i> or <i>portion_bytes</i> has an invalid value.</li><li>• The specified ASSIGN message does not exist.</li><li>• <i>portion</i> does not identify a defined integer part of an ASSIGN message or identifies a nonexistent part.</li></ul>

## SMU\_Assign\_PutText\_

The SMU\_Assign\_PutText\_ function creates or replaces a text part of an ASSIGN message with text obtained from a string variable.

INT PROC SMU_Assign_PutText_(						
		<i>message_number</i> ,				
		<i>portion:portion_bytes</i> ,				
		<i>text:text_bytes</i> )				
		EXTENSIBLE;				
INT		<i>message_number</i> ;	!	in,	required	
STRING	.EXT	<i>portion</i> ;	!	in,	required	
INT		<i>portion_bytes</i> ;	!	in,	required	
STRING	.EXT	<i>text</i> ;	!	in,	required	
INT		<i>text_bytes</i> ;	!	in,	required	TNS,native

*message\_number*

is an integer expression that identifies the ASSIGN message in which to replace *text*. *message\_number* must be a value greater than 0.

*portion*

is the identifier of a message part. *portion* is one of:

- |             |   |
|-------------|---|
| LOGICALNAME | Creates or replaces the logical file name.  |
| TANDEMNAME  | Creates or replaces the Guardian file name. |

*portion\_bytes*

is the size of *portion*, in bytes, and must be a value greater than 0. For TAL, you must supply this value; for COBOL and FORTRAN, the compiler implicitly computes and transmits this value to the function. If *portion\_bytes* exceeds the number of characters in *portion*, the first unused character of *portion* must be a space character.

*text*

is new text for the specified message part. The text format must be appropriate for the message part:

- |             |   |
|-------------|---|
| LOGICALNAME | If a program name is included, the logical file name must be in one of the following formats (maximum 63 characters): |
|-------------|---|

*programname.filename*

*\*.filename*

If program name is omitted, the logical file name must be in the following format (maximum 31 characters):

*filename*

- |            |  |
|------------|--|
| TANDEMNAME | The Guardian file name must be in external format (maximum 34 characters) and can be all blanks. The function does not fill in any missing file name components. |
|------------|--|

*text\_bytes*

is the size of *text*, in bytes, and must have a non-negative value. For TAL, you must supply this value; for COBOL and FORTRAN, the compiler implicitly computes and transmits this value to the function.

If the specified ASSIGN message exists, the function assigns *text* as the specified message part. If the ASSIGN message does not exist and the message part is the logical file name, the function creates an ASSIGN message, assigns *text* as the logical name part, and marks all other message parts as not present. In either case, the function truncates any trailing blanks in the supplied text before assigning it.

Return Value:

- SMU\_Assign\_PutText\_ returns one of the following values:

Return Value	Meaning
0	The operation is successful. <i>text</i> contains a null string.
> 0	The operation is successful. The return value gives the length, in bytes, of the assigned text after blank truncation.
-1	The operation failed because of a logic error, such as: <ul style="list-style-type: none"><li>● <i>message_number</i>, <i>portion_bytes</i>, or <i>text_bytes</i> has an invalid value.</li><li>● The specified ASSIGN message does not exist and either <i>portion</i> is not LOGICALNAME or the new logical file name conflicts with the logical file name of an existing ASSIGN message.</li><li>● <i>portion</i> does not identify a defined textual part of an ASSIGN message or identifies a nonexistent part.</li><li>● <i>text</i> has an invalid value for the specified part.</li></ul>
-2	The operation failed because making changes would invalidate the backup process.
-3	The environment has insufficient allocatable space to complete the operation.

SMU\_Assign\_PutValue\_

The SMU\_Assign\_PutValue\_ function creates or replaces an integer part of an ASSIGN message with the value obtained from an integer variable.

```
INT PROC SMU_Assign_PutValue_ ( message_number,
                                portion:portion_bytes, value )
                                EXTENSIBLE;
    INT      message_number;      ! in,  required
    STRING   .EXT portion;        ! in,  required
    INT      portion_bytes;      ! in,  required
    INT      value;              ! in,  required  TNS,native
```

*message\_number*

is an integer expression that identifies the ASSIGN message in which to replace a value. *message\_number* must have a value greater than 0.

*portion*

is the identifier of the ASSIGN message integer part to create or replace. *portion* is one of:

- ACCESS            Access mode—0 for read-write, 1 for read-only, 2 for write-only, and 3 for extend
- BLKSIZE           Block mode size

EXCLUSION	Exclusion mode—0 for shared, 1 for protected, and 3 for exclusive
FILECODE	File code
PRIEXT	Primary extent size
RECSIZE	Record size
SECEXT	Secondary extent size

*portion\_bytes*

is the size of *portion*, in bytes, and must be a value greater than 0. For TAL, you must supply this value; for COBOL and FORTRAN, the compiler implicitly computes and transmits this value to the function. If *portion\_bytes* exceeds the number of characters in *portion*, the first unused character of *portion* must be a space character.

*value*

is a new value for the message part specified in *portion*. The new value must be appropriate for the message part it creates or replaces.

Return Value

SMU\_Assign\_PutValue\_ returns one of the following values:

Return Value	Meaning
0	The operation is successful.
-1	The operation failed because of a logic error, such as: <ul style="list-style-type: none"><li>• <i>message_number</i> or <i>portion_bytes</i> has an invalid value.</li><li>• The specified ASSIGN message does not exist.</li><li>• <i>portion</i> does not identify a defined integer part of an ASSIGN message.</li><li>• <i>value</i> is invalid for the specified part.</li></ul>
-2	Making changes would invalidate the backup process.
-3	The environment has insufficient allocatable space to complete the operation.

SMU\_Message\_CheckNumber\_

The SMU\_Message\_CheckNumber\_ function determines whether a message exists or reports the number of the highest-numbered saved ASSIGN message.

```
INT PROC SMU_Message_CheckNumber_ ( message_number )
                                     EXTENSIBLE;
    INT message_number;                ! in, required TNS,native
```



*message\_number*

is an integer expression that identifies the message to check. *message\_number* can be one of the following values:

- 3 Checks for the presence of a saved PARAM message
- 1 Checks for the presence of a saved startup message
- 0 Returns the highest message number in the set of saved ASSIGN messages
- >0 Checks for the presence of a saved ASSIGN message

Return Value

SMU\_Message\_CheckNumber\_ returns one of the following values:

Value	Meaning
-3	The specified PARAM message exists.
-1	The specified startup message exists.
0	The specified message does not exist.
> 0	The specified ASSIGN message exists.

SMU\_Param\_Delete\_

The SMU\_Param\_Delete\_ function deletes part or all of a PARAM message.

```
INT PROC SMU_Param_Delete_( portion:portion_bytes )
                                EXTENSIBLE;
    STRING .EXT portion;          ! in, required
    INT      portion_bytes;      ! in, required TNS,native
```

*portion*

specifies what you want to delete. *portion* is either:

- A parameter name Deletes the specified parameter and its associated parameter value.
- \*ALL\* Deletes the entire PARAM message.

*portion\_bytes*

is the size of *portion*, in bytes, and must be a value greater than 0. For TAL, you must supply this value; for COBOL and FORTRAN, the compiler implicitly computes and transmits this value to the function. If *portion\_bytes* exceeds the number of characters in *portion*, the first unused character of *portion* must be a space character.

# Return Value

SMU\_Param\_Delete\_ returns one of the following values:

Return Value	Meaning
0	The operation is successful.
-1	The operation failed because <i>portion-bytes</i> has an invalid value.
-2	The operation failed because making changes would invalidate the backup process.

# SMU\_Param\_GetText\_

The SMU\_Param\_GetText\_ function obtains a part of the PARAM message as text and assigns it to a string variable.

```
INT PROC SMU_Param_GetText_( portion:portion_bytes,
                             text:max_text_bytes )
                             EXTENSIBLE;
  STRING .EXT portion;          ! in,  required
  INT    portion_bytes;        ! in,  required
  STRING .EXT text;            ! out, optional
  INT    max_text_bytes;      ! in,  optional  TNS,native
```

*portion*

specifies what you want to retrieve. *portion* is one of:

- A parameter name               Retrieves the text associated with the specified parameter name (maximum 255 characters).
- \*ALL\*                       Retrieves the entire PARAM message as a string (a sequence of bytes).

*portion\_bytes*

is the size of *portion*, in bytes, and must be a value greater than 0. For TAL, you must supply this value; for COBOL and FORTRAN, the compiler implicitly computes and transmits this value to the function. If *portion\_bytes* exceeds the number of characters in *portion*, the first unused character of *portion* must be a space character.

*text*

is a variable to contain the retrieved text. The function extends the text with blanks on the right side or truncates it as necessary to fit the size of the variable.

*max\_text\_bytes*

is the maximum size of *text*, in bytes, and must be a non-negative value. For TAL, you must supply this value, if *text* is present; for COBOL and FORTRAN, the compiler implicitly computes and transmits this value to the function.

Return Value

SMU\_Param\_GetText\_ returns one of the following values:

Return Value	Meaning
> 0	The operation is successful. The return value gives the length, in bytes, of the retrieved text before truncation or padding.
-1	The operation failed because of a logic error, such as: <ul style="list-style-type: none"><li>• <i>portion_bytes</i> or <i>max_text_bytes</i> has an invalid value.</li><li>• The specified PARAM message does not exist.</li><li>• <i>portion</i> identifies a parameter name that does not exist in the PARAM message.</li></ul>

SMU\_Param\_PutText\_

The SMU\_Param\_PutText\_ function creates or replaces part of a PARAM message.

INT	PROC	SMU_Param_PutText_(	<i>portion</i> : <i>portion_bytes</i> ,	
			<i>text</i> : <i>text_bytes</i> )	
			EXTENSIBLE;	
STRING	.EXT	<i>portion</i> ;	! in,	required
INT		<i>portion_bytes</i> ;	! in,	required
STRING	.EXT	<i>text</i> ;	! in,	optional
INT		<i>text_bytes</i> ;	! in,	optional TNS,native

*portion*

is the name of the parameter you want to create or replace.

*portion\_bytes*

is the size of *portion*, in bytes, and must be a value greater than 0. For TAL, you must supply this value; for COBOL and FORTRAN, the compiler implicitly computes and transmits this value to the function. If *portion\_bytes* exceeds the number of characters in *portion*, the first unused character of *portion* must be a space character.

*text*

is the new text to assign. The function truncates trailing spaces before assigning the text. *text* can have at most 255 characters excluding any trailing blanks.

*text\_bytes*

is the size of *text*, in bytes, and must be a non-negative value. For TAL, you must supply this value, if *text* is present; for COBOL and FORTRAN, the compiler implicitly computes and transmits this value to the function.

If the specified parameter name exists, the function assigns *text* as the associated parameter value. If the parameter name does not exist, the function creates the parameter (or PARAM message if necessary) and assigns *text* as the associated parameter value.

Return Value

SMU\_Param\_PutText\_ returns one of the following values:

Return Value	Meaning
0	The operation is successful. A null string is assigned.
> 0	The operation is successful. The return value gives the length, in bytes, of the assigned text after blank truncation.
-1	The operation failed because of a logic error, such as: <ul style="list-style-type: none"><li>• <i>portion_bytes</i> or <i>text_bytes</i> has an invalid value.</li><li>• <i>text</i> contains more than 255 characters after blank truncation.</li><li>• <i>portion</i> contains a name that does not follow PARAM name rules.</li><li>• The total length of the new PARAM message exceeds the maximum length permitted.</li></ul>
-2	The operation failed because making changes would invalidate the backup process.
-3	The environment has insufficient allocatable space to complete the operation.

SMU\_Startup\_Delete\_

The SMU\_Startup\_Delete\_ function deletes the entire startup message.

```
INT PROC SMU_Startup_Delete_( portion:portion_bytes)
                                EXTENSIBLE;
    STRING .EXT portion;          ! in,  required
    INT    portion_bytes;        ! in,  required  TNS,native
```

*portion*

is the following value:

\*ALL\*      Deletes the entire startup message.

*portion\_bytes*

is the size of *portion*, in bytes, and must be a value greater than 0. For TAL, you must supply this value; for COBOL and FORTRAN, the compiler implicitly computes and transmits this value to the function. If *portion\_bytes* exceeds the number of characters in *portion*, the first unused character of *portion* must be a space character.

You cannot delete parts of a startup message, but you can use `SMU_Startup_PutText_` to set a `STRING` message part to a null string. The resultant message image has an empty parameter string (one with no text) followed by two null characters.

## Return Value

SMU\_Startup\_Delete\_ returns one of the following values:

Return Value	Meaning
0	The operation is successful.
-1	The operation failed because of a logic error, such as: <ul style="list-style-type: none"><li>● <code>portion_bytes</code> has an invalid value.</li><li>● <code>portion</code> does not specify a defined deletable part of the startup message.</li></ul>
-2	The operation failed because making changes would invalidate the backup process.

## SMU\_Startup\_GetText\_

The `SMU_Startup_GetText_` function obtains a specified part of the startup message as text and assigns it to a string variable.

```

INT PROC SMU_Startup_GetText_( portion:portion_bytes,
                               text:max_text_bytes )
                               EXTENSIBLE;
STRING .EXT portion;          ! in,  required
INT     portion_bytes;        ! in,  required
STRING .EXT text;             ! out, optional
INT     max_text_bytes;       ! in,  optional   TNS,native

```

*portion*

identifies what you want to retrieve. *portion* is one of:

IN	Retrieves the IN file name in external format (maximum 34 characters). The file name can be all blanks.
OUT	Retrieves the OUT file name in external format (maximum 34 characters). The file name can be all blanks.
STRING	Retrieves the parameter string, not including trailing null characters (maximum 528 characters).

**VOLUME**      Retrieves the default volume part. If it includes a system name, the text appears as follows (maximum 25 characters):

`\system.$volume.subvolume`

If it omits a system name, the text appears as follows (maximum 17 characters):

`$volume.subvolume`

**\*ALL\***          Retrieves the entire startup message as a string (a sequence of bytes).

*portion\_bytes*

is the size of *portion*, in bytes, and must be a value greater than 0. For TAL, you must supply this value; for COBOL and FORTRAN, the compiler implicitly computes and transmits this value to the function. If *portion\_bytes* exceeds the number of characters in *portion*, the first unused character of *portion* must be a space character.

*text*

is a variable to contain the retrieved text. The function extends the text with blanks on the right side or truncates it as required to match the size of the variable.

*max\_text\_bytes*

is the maximum size of *text*, in bytes, and must be a non-negative value. For TAL, you must supply this value, if *text* is present; for COBOL and FORTRAN, the compiler implicitly computes and transmits this value to the function.

## Return Value

SMU\_Startup\_GetText\_ returns one of the following values:

Return Value	Meaning
> 0	The operation is successful. The return value gives the length, in bytes, of the retrieved text before truncation or padding.
-1	The operation failed because of a logic error, such as: <ul style="list-style-type: none"><li>● <i>portion_bytes</i> or <i>max_text_bytes</i> has an invalid value.</li><li>● The specified startup message does not exist.</li><li>● <i>portion</i> does not identify a defined textual part of the startup message.</li></ul>

## SMU\_Startup\_PutText\_

The `SMU_Startup_PutText_` function creates or replaces a part of the startup message with text obtained from a string variable.

```

INT PROC SMU_Startup_PutText_( portion:portion_bytes,
                               text:text_bytes )
                               EXTENSIBLE;
STRING  .EXT portion;          ! in,  required
INT      portion_bytes;        ! in,  required
STRING  .EXT text;             ! in,  required
INT      text_bytes;           ! in,  required  TNS,native

```

### *portion*

is the name of the startup message part you want to create or replace. *portion* is one of:

IN	IN file name
OUT	OUT file name
STRING	Parameter string
VOLUME	Default volume

### *portion\_bytes*

is the size of *portion*, in bytes, and must be a value greater than 0. For TAL, you must supply this value; for COBOL and FORTRAN, the compiler implicitly computes and transmits this value to the function. If *portion\_bytes* exceeds the number of characters in *portion*, the first unused character of *portion* must be a space character.

### *text*

is the new text to assign to the startup message part. The text must have a format appropriate for the message part. *text* is one of:

IN	The text must be a Guardian file name (maximum 34 characters). The file name can be all blanks. The function does not fill in any missing components.
OUT	The text must be a Guardian file name (maximum 34 characters). The file name can be all blanks. The function does not fill in any missing components.
STRING	The text is the new parameter string, not including trailing null characters (maximum 528 characters).

**VOLUME**    The text is the new default volume part. If it includes a system name, the text must have the following format (maximum 25 characters):

```
\system.$volume.subvolume
```

If it does not include a system name, the text must have the following format (maximum 17 characters):

```
$volume.subvolume
```

*text\_bytes*

is the size of *text*, in bytes, and must be a non-negative value. For TAL, you must supply this value; for COBOL and FORTRAN, the compiler implicitly computes and transmits this value to the function.

If the startup message exists, the function assigns *text* as the message part. If the startup message does not exist, the function creates it, assigns *text* as the message part, and assigns default values to the other message parts. In either case, the function truncates any trailing blanks in the supplied text before assigning it.

**Return Value**

SMU\_Startup\_PutText\_ returns one of the following values:

Return Value	Meaning
0	The operation is successful; a null string is assigned.
> 0	The operation is successful. The return value gives the length, in bytes, of the assigned text after blank truncation.
-1	The operation failed because of a logic error, such as: <ul style="list-style-type: none"><li>• <i>portion_bytes</i> or <i>text_bytes</i> has an invalid value.</li><li>• <i>portion</i> does not identify a defined textual part of the startup message.</li><li>• The text is invalid for the message part (invalid volume, invalid file name, or a parameter string longer than 528 characters).</li></ul>
-2	The operation failed because making changes would invalidate the backup process.
-3	The environment has insufficient allocatable space to complete the operation.

**SMU Function Considerations**

The remainder of this section gives SMU function considerations for COBOL, FORTRAN, TAL, and pTAL.



## COBOL Considerations

You can use the SMU functions in the TNS CRE, in a COBOL run-time environment, or in the native CRE. Rules for using the pre-D20 SMU functions (and the ENV directive) are given in the *COBOL Manual for TNS and TNS/R Programs*. The current SMU functions in this section differ from the pre-D20 SMU functions only in minor ways. These differences are described in the following paragraphs.

You invoke one of the current SMU functions as follows:

```
ENTER [ TAL ] function-name [ OF library-reference ]  
      USING parameter [ , parameter ] ...  
      [GIVING result ]
```

*TAL*

specifies that the function is written in TAL.

*function-name*

is the name of a new SMU function, enclosed in quotation marks; for example:

```
ENTER TAL "SMU_Param_GetText_"
```

*library-reference*

is as described in the COBOL manuals, except that *library-reference* is contained in file CLULIB (which is usually in the \$SYSTEM.SYSTEM subvolume), rather than in a product file such as COBOLLIB. CLULIB contains copies of all pre-D20 and current SMU functions.

*parameter*

is one of the SMU function parameters in this section. (The *cplist* parameter of pre-D20 SMU functions is not a part of the current SMU functions.)

*portion*

is an alphanumeric data item that specifies the message part you want to retrieve or change. The compiler implicitly computes and transmits the size, in bytes, of *portion* to the function.

*text*

is an alphanumeric data item for retrieving text or for specifying new text. The compiler implicitly computes and transmits the size, in bytes, of *text* to the function. Any JUSTIFIED clause of the data item is ignored, and text is truncated or extended to fit the data item.

*value*

is a NATIVE-2 data item or a COMPUTATIONAL numeric data item described with one-to-four 9s preceded by an S; for example:

```
PICTURE S9(4)
```

*message\_number*

is any actual COBOL parameter that evaluates to a number in the range -32768 through 32767, as permitted by a particular function.

*message\_number* can be a COMPUTATIONAL, DISPLAY, or NATIVE numeric data item, a numeric literal, or an expression enclosed in parentheses. If the parameter does not mathematically evaluate to an integer, the compiler changes the result into an integer value. To specify conversion options, you can use COBOL functions INTEGER and INTEGER\_PART.

*name*

is an alphanumeric data item that specifies a logical file name. The compiler implicitly computes and transmits the size, in bytes, of *name* to the function.

*result*

is a numeric data item as described in the *COBOL Manual for TNS and TNS/R Programs* for the pre-D20 SMU functions.

## COBOL Examples

The following pairs of examples show how each new SMU function corresponds to a pre-D20 SMU function:

```
ENTER TAL "SMU_Assign_Delete_"
  USING message-number, portion          GIVING result
ENTER "DeleteAssign"
  USING portion, cplist, message-number  GIVING result

ENTER TAL "SMU_Assign_GetText_"
  USING message-number, portion, text    GIVING result
ENTER "GetAssignText"
  USING portion, text, message-number    GIVING result

ENTER TAL "SMU_Assign_GetValue_"
  USING message-number, portion, value   GIVING result
ENTER "GetAssignValue"
  USING portion, value, message-number   GIVING result

ENTER TAL "SMU_Assign_PutText_"
  USING message-number, portion, text    GIVING result
ENTER "PutAssignText"
  USING portion, text, cplist, message-number  GIVING result

ENTER TAL "SMU_Assign_PutValue_"
  USING message-number, portion, value   GIVING result
```

```

ENTER "PutAssignValue"
  USING portion, value, cplist, message-number  GIVING result

ENTER TAL "SMU_Param_Delete_"  USING portion  GIVING result
ENTER "DeleteParam"  USING portion, cplist  GIVING result

ENTER TAL "SMU_Param_GetText_"
  USING portion, text  GIVING result
ENTER "GetParamText"
  USING portion, text  GIVING result

ENTER TAL "SMU_Param_PutText_"
  USING portion, text  GIVING result
ENTER "PutParamText"
  USING portion, text, cplist  GIVING result

ENTER TAL "SMU_Startup_Delete_"  USING portion  GIVING result
ENTER "DeleteStartup"  USING portion, cplist  GIVING result

ENTER TAL "SMU_Startup_GetText_"
  USING portion, text  GIVING result
ENTER "GetStartupText"
  USING portion, text  GIVING result

ENTER TAL "SMU_Startup_PutText_"
  USING portion, text  GIVING result
ENTER "PutStartupText"
  USING portion, text, cplist  GIVING result

ENTER TAL "SMU_Assign_CheckName_"  USING name  GIVING result
ENTER "CheckLogicalName"  USING name  GIVING result

ENTER TAL "SMU_Message_CheckNumber_"
  USING message-number  GIVING result
ENTER "CheckMessage"
  USING message-number  GIVING result

```

## FORTRAN Considerations

You can use the SMU functions in either the TNS CRE or in a FORTRAN run-time environment. Rules for using the pre-D20 SMU functions (and the ENV directive) are given in the *FORTRAN Reference Manual*. The current SMU functions in this section differ from the pre-D20 SMU functions only in minor ways. These differences are described in the following paragraphs.

Before you reference a current SMU function, you must declare it in a GUARDIAN or CONSULT directive that specifies an object file that contains a copy of the function. The current SMU functions are contained in CLULIB (usually in \$SYSTEM.SYSTEM), rather than in a FORTRAN product file. CLULIB also contains all the pre-D20 SMU functions.

You invoke the current SMU functions as follows:

```
result = functionname ( parameter [ , parameter ] ... )
```

*result*

is an integer variable as described in the *FORTRAN Reference Manual* for the old SMU functions.

*functionname*

is the name of a current SMU function.

*parameter*

is one of the SMU function parameters in this section. (The *cplist* parameter of pre-D20 SMU functions is not a part of the new SMU functions.)

*portion*

is a character expression that specifies the message part you want to retrieve or change. The compiler implicitly computes and transmits the size, in bytes, of *portion* to the function.

*text*

is a character variable for new or retrieved text. The compiler implicitly computes and transmits the size, in bytes, of *text* to the function.

*value*

is an integer variable.

*message\_number*

is an integer expression that evaluates to a number in the range -32768 through 32767, as permitted by a particular function.

*name*

is character expression that specifies a logical file name. The compiler implicitly computes and transmits the size, in bytes, of *name* to the function.

## FORTRAN Examples

The following pairs of examples show how each current SMU function corresponds to a pre-D20 SMU function:

```
result = SMU_Assign_Delete_ (messagenumber, portion)
result = DeleteAssign (portion, cplist, messagenumber)
```

```
result = SMU_Assign_GetText_ (messagenumber, portion, text)
result = GetAssignText (portion, text, messagenumber)
```

```

result = SMU_Assign_GetValue_ (messagenumber, portion, value)
result = GetAssignValue (portion, value, messagenumber)

result = SMU_Assign_PutText_ (messagenumber, portion, text)
result = PutAssignText (portion, text, cplist, messagenumber)

result = SMU_Assign_PutValue_ (messagenumber, portion, value)
result = PutAssignValue (portion, value, cplist,
                        messagenumber)

result = SMU_Param_Delete_ (portion)
result = DeleteParam (portion, cplist)

result = SMU_Param_GetText_ (portion, text)
result = GetParamText (portion, text)

result = SMU_Param_PutText_ (portion, text)
result = PutParamText (portion, text, cplist)

result = SMU_Startup_Delete_ (portion)
result = DeleteStartup (portion, cplist)

result = SMU_Startup_GetText_ (portion, text)
result = GetStartupText (portion, text)

result = SMU_Startup_PutText_ (portion, text)
result = PutStartupText (portion, text, cplist)

result = SMU_Assign_CheckName_ (name)
result = CheckLogicalName (name)

result = SMU_Message_CheckNumber_ (messagenumber)
result = CheckMessage (messagenumber)

```

## TAL Considerations

TAL routines can use the new SMU functions if the TAL routines are part of a program that can run in the TNS CRE or part of a program that can run in a COBOL or FORTRAN run-time environment. TAL routines cannot use the old SMU functions.

The file CLUDECS, which is usually in the \$SYSTEM.SYSTEM subvolume, contains the new SMU function declarations.

All other usage considerations applicable to TAL are included in the function descriptions in this section.

## EpTAL Considerations

EpTAL-compiled routines can use the current SMU functions if the EpTAL routines are part of a program that can run in the TNS/E native CRE (for example, a program that has its main routine written in C instead of pTAL). EpTAL-compiled routines cannot use the pre-D20 SMU functions.

The file CLURDECS, which is usually in the \$SYSTEM.SYSTEM subvolume, contains the current SMU function declarations.

All other usage considerations applicable to EpTAL are included in the function descriptions in this section.

## pTAL Considerations

pTAL routines can use the current SMU functions if the pTAL routines are part of a program that can run in the TNS/R native CRE (for example, a program that has its main routine written in C instead of pTAL). pTAL routines cannot use the pre-D20 SMU functions.

The file CLURDECS, which is usually in the \$SYSTEM.SYSTEM subvolume, contains the current SMU function declarations.

All other usage considerations applicable to pTAL are included in the function descriptions in this section.

# Run-Time Diagnostic Messages

This section lists in numerical order the run-time diagnostic messages that you can see in your log file when you run a program that uses the Common Run-Time Environment (CRE). This section covers the following topics:

- [Error Effects and Recovery](#) on page 10-1
- [Format of Messages in This Section](#) on page 10-2
- [Trap and Signal Messages](#) on page 10-3
- [CRE Service Function Messages](#) on page 10-6
- [Heap-Management Messages](#) on page 10-12
- [Math Function Messages](#) on page 10-15
- [Function Parameter Messages](#) on page 10-17
- [Input/Output Messages](#) on page 10-18
- [COBOL Messages](#) on page 10-25
- [FORTRAN Messages](#) on page 10-25
- [Native CRE Messages](#) on page 10-25
- [Mapping Message Numbers Between Run-Time Environments](#) on page 10-26

The messages in this section are grouped according to logical functions. Messages associated with math functions are in a subsection, FORTRAN messages are in a separate subsection, and so forth. However, any routine or run-time library in your program can use any of the messages in this section. Thus, math function messages can be generated by routines that are not math functions but for which the message text is applicable.

The language-specific messages for COBOL and FORTRAN are current as of the time this manual was released. If you cannot find a particular error message, refer to the appropriate language reference manual or softdoc.

## Error Effects and Recovery

The CRE is the only part of a process that invokes system procedures to report the messages described in this section. The CRE reports messages for either of the following reasons:

- It detected an error while performing its own tasks as a resource manager.
- A routine in a program or run-time library requested that it display a message.

For messages that the CRE reports on its own behalf, the descriptions of error effects and possible recovery methods are as descriptive as possible. For more details, refer to other sections of this manual or to one or more of the following manuals:

- *Guardian Programmer's Guide*
- *Open Systems Services Programmer's Guide*
- *Guardian Procedure Calls Reference Manual*
- *TACL Reference Manual*

- *Spooler Programmer's Guide*

For messages that the CRE reports on behalf of routines that call it, the descriptions are less precise because the effect on your program and possible recovery actions are language and application dependent. For more information on errors that are reported by individual languages, refer to the language reference manual for the routine that caused the error.

## Format of Messages in This Section

This subsection describes the format of the messages the CRE writes to the log file. Elements surrounded by square brackets (“[” and “]”) are not included in all messages.

```

process_name - *** Run-time Error nnn ***
process_name - message [ ( additional_information ) ]
[ process_name - optional_text ]
process_name - From: top_of_stack
process_name -      :           :           :
process_name -      bottom_of_stack

```

*process\_name*

identifies the process in which the error occurred.

*nnn*

is the CRE message number of the message described in this section.

*message*

is the text associated with message number *nnn*.

*additional\_information*

if present, gives more detail about *message*. For example, if an error occurs when accessing a file, *additional\_information* might be the file-system error number for the error that occurred.

*optional\_text*

if present, provides additional information about the error. For example, it might show the FORTRAN unit number, or the state of a COBOL file: open, closed, and so forth. The *optional\_text* helps you identify the cause of the error message.

*top\_of\_stack*

shows the name of the procedure that invoked the run-time library routine in which the error was detected, the offset within the procedure, and the number of the code segment in which the procedure's code is located.



*bottom\_of\_stack*

shows the name of the first procedure—the main procedure—of the process in which the error occurred, the offset within the procedure, and the number of the code segment in which procedure's code is located. The stack trace includes all procedures between *top\_of\_stack* and *bottom\_of\_stack*.

The following examples show messages that the CRE might write to the log file:

- If a program passes a negative value to a square root function, the CRE writes a message such as the following to the log file:

```
\NODE.$Z012:3 - *** Run-time Error 049 ***
\node.$Z012:3 - Square root domain fault
\node.$Z012:3 - From: DRAWIT + %513, UC.00
\node.$Z012:3 -          CIRCLE + %21, UC.00
\node.$Z012:3 -          MYPROG + %7, UC.00
```

- If a program tries to open standard input but the file does not exist, the CRE writes a message such as the following to the log file:

```
\NODE.$Z012:3 - *** Run-time Error 059 ***
\node.$Z012:3 - Standard input file error (11)
\node.$Z012:3 - From: READREC + %54, UC.00
\node.$Z012:3 -          NEXTREC + %15, UC.00
\node.$Z012:3 -          COMPUTE + %214, UC.00
\node.$Z012:3 -          MYPROG + %7, UC.00
```

Note that the error message includes the number of the file-system error number (11).

- If a COBOL program cannot create a new file, the CRE writes a message such as the following to the log file. Note that the message includes an informational line that shows the FD-name of the file, the external Guardian file name, and the file's status (closed):

```
\NODE.$Z012:3 - *** Run-time Error 153 ***
\node.$Z012:3 - Create of new file failed with error 153
\node.$Z012:3 - File FD-IN-FILE = $vol.subvol.file, closed
\node.$Z012:3 - From: NEWFILE + %66, UC.00
\node.$Z012:3 -          PM-ADMIN + %71, UC.00
\node.$Z012:3 -          MYPROG + %7, UC.00
```

## Trap and Signal Messages

The TNS CRE reports the messages in this subsection if a trap occurs and your program has neither disabled traps nor enabled its own trap handler. The native CRE reports the messages in this subsection if a signal is raised and your program does not have its own signal handler for the signal that was raised. The CRE or run-time library terminates your program.

The CRE treats trap 4, arithmetic fault, as a program logic error, not a trap. If your program has disabled overflow traps, the TNS CRE returns control to your run-time library. The native CRE raises a `SIGFPE` signal instead. See the language-specific

manuals for the routines in your program for additional detail. The *Guardian Programmer's Guide* describes traps and signals.

## 1

Unknown trap

The CRE trap processing function was called with an unknown trap number.

## 2

Illegal address reference

An address was specified that was not within either the virtual code area or the virtual data area allocated to the process.

## 3

Instruction failure

An attempt was made to:

- Execute a code word that is not an instruction.
- Execute a privileged instruction by a nonprivileged process.
- Reference an illegal extended address.

## 4

Arithmetic fault

In the TNS environment, the overflow bit in the environment-register, ENV.<10>, was set to 1. In the native environment, a SIGFPE was raised. In either environment, the fault occurs for one of the following reasons:

- The result of a signed arithmetic operation could not be represented with the number of bits available for the particular data type.
- An division operation was attempted with a zero divisor.

## 5

Stack overflow

A stack overflow fault occurs if:

- An attempt was made to execute a procedure or subprocedure whose local or sublocal data area extends into the upper 32K of the user data area.

- There was not enough remaining virtual data space for a system procedure to execute.
- The native environment exceeded the maximum stack space available.

The amount of virtual data space available is G[0] through G[32767].

System procedures require approximately 350 words of user-data stack space to execute.

## 6

Process loop-timer timeout

The new time limit specified in the latest call to SETLOOPTIMER has expired.

## 7

Memory manager read error

An unrecoverable, read error occurred while the program was trying to bring in a page from virtual memory.

## 8

Not enough physical memory

This fault occurs for one of the following reasons:

- A page fault occurred, but there were no physical memory pages available for overlay.
- Disk space could not be allocated while the program is using extensible segments.

## 9

Uncorrectable memory error

An uncorrectable memory error occurred.

## 10

Interface limit exceeded

The operating system reported a trap 5.

# CRE Service Function Messages

The CRE writes the messages in this subsection if an error occurs during its own processing or if it receives a request from a run-time library to report a specific message.

## 11

Corrupted environment
-----------------------

**Cause.** CRE or run-time library data is invalid.

**Effect.** The CRE invokes PROCESS\_STOP\_, specifying the ABEND variant and the text “Corrupted environment.”

**Recovery.** In the TNS environment, the program might have written data in the upper 32K words of the user data segment. The upper 32K words are reserved for TNS CRE and run-time library data.

In the native environment, the run-time environment has been corrupted. You might have written data over run-time data structures. |

Check the program's logic. Use Inspect, Native Inspect, or Visual Inspect to help isolate the problem or consult your system administrator. |

## 12

Logic error
-------------

**Cause.** The CRE or a run-time library detected a logic error within its own domain. For example, although each data item it is using is valid, the values of the data items are mutually inconsistent.

**Effect.** The CRE invokes PROCESS\_STOP\_, specifying the ABEND variant and the text “Logic error.”

**Recovery.** In the TNS environment, the program might have written data in the upper 32K words of the user data segment. The upper 32K words are reserved for TNS CRE and run-time library data.

In the native environment, the run-time environment has been corrupted. You might have written data over run-time data structures. |

Check the program's logic. Use Inspect, Native Inspect, or Visual Inspect to help isolate the problem or consult your system administrator. |

## 13

MCB pointer corrupt

**Cause.** The pointer at location G[0] of the program's user data segment to its primary data structure—the Master Control Block (MCB)—does not point to the MCB. Both the CRE and run-time libraries can report this error.

In the native environment, this error can occur if the MCB run-time data structure has been corrupted.

**Effect.** The CRE attempts to restore the pointer at G[0] and to write a message to the standard log file. However, because its environment might be corrupted, the CRE might not be able to log a message. In that case, it calls `PROCESS_STOP_`, specifying the ABEND variant, and the text "Corrupted Environment".

**Recovery.** Check the program's logic to see if it overwrote the MCB pointer at G[0]. Use a symbolic debugger appropriate for the type of code to help isolate the problem. See [Using the Inspect, Native Inspect, and Visual Inspect Symbolic Debuggers With CRE Programs](#) on page 2-62 for details of how to determine where the program overwrites G[0].

## 14

Premature takeover

**Cause.** The CRE backup process received an operating system message that it had become the primary process but it had not yet received all of its initial checkpoint information from its predecessor primary process.

**Effect.** The CRE invokes `PROCESS_STOP_`, specifying the ABEND variant and the text "Premature takeover."

**Recovery.** If the takeover occurred because of faulty program logic, correct the program's logic. If the takeover occurred for other reasons, such as a hardware failure, you might want to rerun the program. Do not rerun the program if doing so will duplicate operations already performed, such as updating a database a second time.

## 15

Checkpoint list inconsistent

**Cause.** A list of checkpoint item descriptors that the CRE maintains for process pairs was invalid.

**Effect.** The CRE terminates the program.

**Recovery.** The list of items to checkpoint is maintained in the program's address space. Check the program's logic. The program might have overwritten the checkpoint

list. Use a symbolic debugger appropriate for the type of code to help isolate the problem.

## 16

Checkpoint list exhausted

**Cause.** The CRE did not have enough room to store all of the checkpoint information required by the program.

**Effect.** Program behavior is language and application dependent.

**Recovery.** Increase the checkpoint list object's size. See the language manual for the routine that allocates your checkpoint list.

## 17

Cannot obtain control space

**Cause.** The CRE or a run-time library could not obtain heap space for all of its data.

**Effect.** If the request came from the CRE, it terminates the program. Otherwise, program behavior is language and application dependent.

**Recovery.** You might be able to increase the amount of control space available to your program by reducing the number of files your program has open at the same time or by decreasing the size of buffers allocated to open files.

## 18

Extended Stack Overflow

**Cause.** A module could not obtain sufficient extended stack space for its local data.

**Effect.** Program behavior is language and application dependent.

**Recovery.** Increase the extended stack's size. See the language manual for the routine that caused the extended stack overflow for details on increasing the size of the extended stack.

## 20

Cannot utilize filename

**Cause.** A string, expected to be a valid file name, could not be manipulated as a Guardian external file name.

**Effect.** If the file name came from the CRE, the program is terminated. Otherwise, program behavior is language and application dependent.

**Recovery.** Check that the file names in the program are valid Guardian file names.

## 21

```
Cannot read initialization messages ( error )
```

**Cause.** During program initialization, the CRE could not read all of the messages (start-up message, PARAM message, ASSIGN messages, and so forth) it expected from the file system. *error* is the file-system error number the CRE received when it couldn't read an initialization message.

**Effect.** The CRE terminates the program.

**Recovery.** Consult your system administrator.

## 22

```
Cannot obtain program filename
```

**Cause.** The CRE could not obtain the name of the program file from the operating system.

**Effect.** The CRE terminates the program.

**Recovery.** Consult your system administrator.

## 23

```
Cannot determine filename ( error )  
program_name.logical_name
```

**Cause.** The CRE could not determine the physical file name associated with *program\_name.logical\_name*.

**Effect.** The CRE terminates the program.

**Recovery.** Correct the *program\_name.logical\_name* and rerun your program. See the HP *Tandem Advanced Command Language (TACL) Reference Manual* for general information on ASSIGN commands. See the reference manual for your program's main routine for more information on using ASSIGNS.

## 24

```
Conflict in application of ASSIGN
program_name.logical_name
```

**Cause.** ASSIGN values in your TACL environment conflict with each other. For example:

```
ASSIGN    A, $B1.C.D
ASSIGN *.A, $B2.C.D
```

The first ASSIGN specifies that the logical name A can appear in no more than one program file. The second assign specifies that the name A can appear in an arbitrary number of program files. The CRE cannot determine whether to use the file C.D on volume \$B1 or on volume \$B2.

**Effect.** The CRE terminates the program.

**Recovery.** Correct the ASSIGNS in your TACL environment. See the HP *Tandem Advanced Command Language (TACL) Reference Manual* for more information on using ASSIGNS.

## 25

```
Ambiguity in application of ASSIGN
logical_name
```

**Cause.** Your TACL environment specifies an ASSIGN such as:

```
ASSIGN    A, $B1.C.D
```

but the program contains more than one logical file named A.

**Effect.** The CRE terminates the program.

**Recovery.** Correct the ASSIGNS in your TACL environment. See the HP *Tandem Advanced Command Language (TACL) Reference Manual* for more information on using ASSIGNS.

## 26

```
Invalid PARAM value text ( error )
PARAM name 'value'
```

**Cause.** A PARAM specifies a value that is not defined by the CRE. For example, the value for a DEBUG PARAM must be either ON or OFF:

```
PARAM DEBUG [ ON  ]
             [ OFF ]
```

The CRE reports this error if a DEBUG PARAM has a value other than ON or OFF. *error*, if present, is a file-system error.



**Effect.** The CRE terminates the program.

**Recovery.** Modify the PARAM text and rerun your program. See the HP *Tandem Advanced Command Language (TACL) Reference Manual* for more information on using PARAMs.

## 27

```
Ambiguity in application of PARAM  
PARAM name 'value'
```

**Cause.** A PARAM specifies a value that is ambiguous in the current context. For example, the PARAM specification:

```
PARAM PRINTER-CONTROL A
```

is ambiguous if the program contains more than one logical file named A.

**Effect.** The CRE terminates the program.

**Recovery.** Correct the PARAM in your TACL environment. See the HP *Tandem Advanced Command Language (TACL) Reference Manual* for more information on using PARAMs.

## 28

```
Missing language run-time library -- language
```

**Cause.** The run-time library for a module that is written in *language* is not available to the program.

**Effect.** The CRE terminates the program.

**Recovery.** Consult your system administrator.

## 29

```
Program incompatible with run-time library -- language
```

**Cause.** The *language* compiler used features that are not supported by the *language* run-time library that the program used.

**Effect.** The CRE terminates the program.

**Recovery.** Use a compiler and run-time library that are compatible. You might need to consult your system administrator.

# Heap-Management Messages

The CRE or run-time libraries report the messages in this subsection if they detect an error while accessing the heap.

## 30

Unknown heap error

**Cause.** A heap-management routine was called with an invalid error number.

**Effect.** The CRE terminates the program.

**Recovery.** Refer to the reference manual that corresponds to the language in which the routine that requests heap space is written. You might need to consult your system administrator.

## 31

Cannot obtain data space

**Cause.** Your program, or the run-time library for one of the modules in your program, requested more space on the heap than is currently available.

**Effect.** Program behavior is language and application dependent.

**Recovery.** Refer to the reference manual that corresponds to the language in which the routine that requests heap space is written. You might need to consult your system administrator.

## 32

Invalid heap or heap control block  
Process heap size is 0

**Cause.** The CRE or a run-time library found invalid data in the user heap or in the heap control block.

Your program might be writing information over the heap or heap control block. An invalid pointer or indexing operation could cause this error.

“Process heap size is 0” appears when your program or a run-time library requested space, but the process has no user heap.

**Effect.** Program behavior is language and application dependent.

**Recovery.** In the TNS environment, the program might have written data in the upper 32K words of the user data segment or in the extended segment. The upper 32K words of the user data area are reserved for CRE and run-time library data. In a small-

memory model, the heap is allocated in the lower 32K words of the user data segment. In a large memory model, the heap is allocated in an extended memory segment. Check the program's logic. Use Inspect or Visual Inspect to help isolate the problem or consult your system administrator.

If this error occurs in the native environment, use Native Inspect or Visual Inspect to help isolate the problem. In the case of "Process heap size is 0," specify the existence of a user heap when building your program.

### 33

```
Released space address is 0
```

**Cause.** The address of a block of memory to return to the heap was zero.

**Effect.** Program behavior is language and application dependent.

**Recovery.** Correct the program to pass the correct address.

### 34

```
Released space not allocated
```

**Cause.** The address of a block of memory to return to the heap did not point to a block allocated from the heap.

**Effect.** Program behavior is language and application dependent.

**Recovery.** Correct the program to return the correct pointer value.

### 35

```
Heap corrupted ( 32-bit_octal_address )
```

**Cause.** The CRE or a run-time library found invalid information at location *32-bit\_octal\_address* of the heap.

**Effect.** Program behavior is language and application dependent.

**Recovery.** The program might have written data over the heap. In a small-memory model, the heap is allocated in the lower 32K words of the user data segment, just below the run-time stack. In a large memory model, the heap is allocated in an extended memory segment. Check the program's logic. Use Inspect or Visual Inspect to help isolate the problem or consult your system administrator.

If this error occurs in the native environment, check the program's logic. Use Native Inspect or Visual Inspect to help isolate the problem or consult your system administrator.

## 36

Invalid operation for heap

**Cause.** The request you made is not compatible with the heap that you referenced.

**Effect.** Program behavior is language and application dependent.

**Recovery.** Consult your system administrator.

## 37

Mark address or space corrupt

**Cause.** An address passed as a heap marker does not point to a mark.

**Effect.** Program behavior is language and application dependent.

**Recovery.** In the TNS environment, ensure that the program passed the correct address of a mark. If it did, the heap might be corrupted. Check the program's logic. CRE and run-time library data is stored in the upper 32K words of the user data segment and in the primary extended data segment. The program might have overwritten CRE or run-time library data. Use Inspect or Visual Inspect to help isolate the problem. You might need to consult your system administrator.

If this error occurs in the native environment, check the program's logic. Use Native Inspect or Visual Inspect to help isolate the problem or consult your system administrator.

# Function Parameter Message

Run-time libraries report the message in this subsection if an error is detected in a parameter passed to a function.

## 40

Invalid function parameter

**Cause.** A function detected a problem with its parameters.

**Effect.** Program behavior is language and application dependent.

**Recovery.** Correct the parameter you are passing.

# Math Function Messages

Run-time libraries report the messages in this subsection if an error is detected in a math function.

## 41

Range fault

**Cause.** An arithmetic overflow or underflow occurred while evaluating an arithmetic function.

**Effect.** Program behavior is language and application dependent.

**Recovery.** Modify the program to pass values to the arithmetic functions that do not cause overflow.

## 42

Arccos domain fault

**Cause.** The parameter passed to the arccos function was not in the range:

$-1.0$  less than or equal to *parameter* less than or equal to  $1.0$

**Effect.** Program behavior is language and application dependent.

**Recovery.** Modify the program to pass a valid value to the arccos function.

## 43

Arcsin domain fault

**Cause.** The parameter passed to the arcsin function was not in the range:

$-1.0$  less than or equal to *parameter* less than or equal to  $1.0$

**Effect.** Program behavior is language and application dependent.

**Recovery.** Modify the program to pass a valid value to the arcsin function.

## 44

Arctan domain fault

**Cause.** Both of the parameters to an arctan2 function were zero. At least one of the parameters must be nonzero.

**Effect.** Program behavior is language and application dependent.

**Recovery.** Modify the program to pass the correct value to the arctan2 function.

## 46

Logarithm function domain fault

**Cause.** The parameter passed to a logarithm function was less than or equal to zero. The parameter to a logarithm function must be greater than zero.

**Effect.** Program behavior is language and application dependent.

**Recovery.** Modify the program to pass a valid value to the logarithm function.

## 47

Modulo function domain fault

**Cause.** The value of the second parameter to a modulo function was zero. The second parameter to a modulo function must be nonzero.

**Effect.** Program behavior is language and application dependent.

**Recovery.** Modify the program to pass a nonzero value to the modulo function.

## 48

Exponentiation domain fault

**Cause.** Parameters to a Power function were not acceptable. Given the expression

$x^y$

the following parameter combinations produce this message:

$x = 0$  and  $y$  is less than or equal to 0  
 $x < 0$  and  $y$  is not an integral value

**Effect.** Program behavior is language and application dependent.

**Recovery.** Modify the program to pass values that do not violate the above combinations.

## 49

Square root domain fault

**Cause.** The parameter to a square root function was a negative number. The parameter must be greater than or equal to zero.

**Effect.** Program behavior is language and application dependent.

**Recovery.** Modify the program to pass a nonnegative value to the square root function.

# Function Parameter Messages

The CRE or run-time libraries report the messages in this subsection if there is a problem with the parameters passed to a function.

## 55

Missing or invalid parameter
------------------------------

**Cause.** A required parameter is missing or too many parameters were passed.

**Effect.** Program behavior depends on the function that was called and the language in which it is written.

**Recovery.** Correct the program to pass a valid parameter.

## 56

Invalid parameter value
-------------------------

**Cause.** The value passed as a procedure parameter was invalid.

**Effect.** Program behavior depends on the function that was called and the language in which it is written.

**Recovery.** Correct the program to pass a valid parameter value.

## 57

Parameter value not accepted
------------------------------

**Cause.** The value passed as a procedure parameter is not acceptable in the context in which it is passed. For example, the number of bytes in a write request is greater than the number of bytes per record in the file.

**Effect.** Program behavior depends on the function that was called and the language in which it is written.

**Recovery.** Correct the program to pass a valid parameter.

# Input/Output Messages

The CRE or run-time libraries report the messages in this subsection if an error occurs when calling an I/O function.

## 59

```
Standard input file error ( error )  
Unable to open <filename>
```

**Cause.** The file system reported an error when a routine tried to access the standard input file. *error* is a file-system error number.

“Unable to open <filename>” appears when the C run-time library cannot open the standard input file during program initialization. <filename> shows the name for which the open operation failed.

**Effect.** The CRE can report this error when it closes your input file. All other instances are language and application dependent.

**Recovery.** If the error was caused by a read request from your program, correct your program. You might need to ensure that your program handles conditions that are beyond your control such as losing a path to the device. Also refer to error handling in this manual and in the language manual for the routine in your program that detected the error.

If the error was caused by a read request from the CRE, consult your system administrator.

If the error was caused during program initialization, specify an acceptable input file when executing your program.

## 60

```
Standard output file error ( error )  
Unable to open <filename>
```

**Cause.** The file system reported an error when the CRE called a file system procedure to access standard output. *error* is the file-system error number.

“Unable to open <filename>” appears when the C run-time library cannot open the standard output file during program initialization. <filename> shows the name for which the open operation failed.

**Effect.** The CRE can report this error when it closes your output file. All other instances are language and application dependent.

**Recovery.** If the error was caused by a write request from your program, correct your program. You might need to ensure that your program handles conditions that are beyond your control such as losing a path to the device. Also refer to error handling in



this manual and in the language manual for the routine in your program that detected the error.

If the error was caused by a write request from the CRE, consult your system administrator.

If the error was caused during program initialization, specify an acceptable output file when executing your program.

## 61

```
Standard log file error ( error )  
Unable to open filename
```

**Cause.** The file system reported an error when the CRE called a file system procedure to access the standard log file. *error* is the file system error number.

“Unable to open *filename*” appears when the C run-time library cannot open the standard log file during program initialization. *filename* shows the name for which the open operation failed.

**Effect.** The CRE terminates your program.

**Recovery.** Consult your system administrator.

If the error was caused during program initialization, specify an acceptable log file when executing your program.

## 62

```
Invalid GUARDIAN file number
```

**Cause.** A value that is expected to be a Guardian file number is not the number of an open file.

**Effect.** Program behavior is language and application dependent.

**Recovery.** Consult your system administrator.

## 63

```
Undefined shared file
```

**Cause.** A parameter was not the number of a shared (standard) file where one was expected.

**Effect.** Program behavior is language and application dependent.

**Recovery.** Consult your system administrator.

## 64

File not open

**Cause.** A request to open a file failed because the file device is not supported.

**Effect.** Program behavior is language and application dependent.

**Recovery.** Consult your system administrator.

## 65

Invalid attribute value

**Cause.** A parameter to an open operation was not a meaningful value. For example, the `CRE_File_Open_sync_receive_depth` parameter must be a nonnegative number. This message might be reported if the `sync_receive_depth` parameter is negative.

**Effect.** Program behavior is language and application dependent.

**Recovery.** Consult your system administrator.

## 66

Unsupported file device

**Cause.** The CRE received a request to access a device that it does not support.

**Effect.** Program behavior is language and application dependent.

**Recovery.** Consult your system administrator.

## 67

Access mode not accepted

**Cause.** The value of the `access` parameter to an open operation was not valid in the context in which it was used. For example, it is invalid to open a spool file for input.

**Effect.** Program behavior is language and application dependent.

**Recovery.** Consult your system administrator.

## 68

Nowait value not accepted

**Cause.** The value of the *no\_wait* parameter to an open operation was not valid in the context in which it was used. For example, it is invalid to specify a nonzero value for *no\_wait* for a device that does not support nowait operations.

**Effect.** Program behavior is language and application dependent.

**Recovery.** Consult your system administrator.

## 69

Syncdepth not accepted

**Cause.** The value of the *sync\_receive\_depth* parameter to an open operation was not valid in the context in which it was used. For example, it is not valid to specify a *sync\_receive\_depth* greater than one for a standard file.

**Effect.** Program behavior is language and application dependent.

**Recovery.** Consult your system administrator.

## 70

Options not accepted

**Cause.** The value of an open operation *options* parameter was not valid in the context in which it was used.

**Effect.** Program behavior is language and application dependent.

**Recovery.** Consult your system administrator.

## 71

Inconsistent attribute value

**Cause.** A routine requested a connection to a standard file that was already open, and the attributes of the new open request conflict with the attributes specified when the file was first opened.

**Effect.** Program behavior is language and application dependent.

**Recovery.** If your program supplied the attribute values, correct and rerun your program. Otherwise, consult your system administrator.

## 75

```
Cannot obtain buffer space
```

**Cause.** A routine was not able to obtain buffer space.

**Effect.** Program behavior is language and application dependent.

**Recovery.** Program recovery is language and application dependent.

## 76

```
Invalid external filename ( error )
```

**Cause.** A value that was expected to be a Guardian external file name is not in the correct format.

**Effect.** Program behavior is language and application dependent.

**Recovery.** If you supplied an invalid file name, correct the file name and rerun your program. Otherwise, consult your system administrator.

## 77

```
EDITREADINIT failed ( error )
```

**Cause.** A call to EDITREADINIT failed. *error*, if present, gives the reason for the failure. Possible values of *error* are:

**Effect.** Program behavior is language and application dependent. See the *Guardian Procedure Calls Reference Manual* for more information.

**Recovery.** Recovery is language and application dependent.

## 78

```
EDITREAD failed ( error )
```

**Cause.** A call to EDITREAD failed. *error*, if present, gives the reason for the failure. Possible values of *error* are:

**Effect.** Program behavior is language and application dependent.

**Recovery.** See the *Guardian Procedure Calls Reference Manual* for more information.

## 79

```
OpenEdit failed ( error )
```

**Cause.** A call to OPENEDIT\_ failed. *error*, if present, is the error returned by OPENEDIT\_. A negative number is a format error. A positive number is a file-system error number.

**Effect.** Program behavior is language and application dependent.

**Recovery.** See the *Guardian Procedure Calls Reference Manual* for more information.

## 80

```
Spooler initialization failed ( error )
```

**Cause.** An initialization operation to a spooler collector failed. *error*, if present, is the file-system error number returned by the SPOOLSTART system procedure.

**Effect.** Program behavior is language and application dependent.

**Recovery.** See the *Spooler Programmer's Guide* for more information.

## 81

```
End of file
```

**Cause.** A routine detected an end-of-file condition.

**Effect.** Program behavior is language and application dependent.

**Recovery.** Correct your program to allow for an end-of-file condition or ensure that your program can determine when all of the data has been read.

## 82

```
Guardian I/O error nnn
```

**Cause.** An operating system routine returned file-system error *nnn*. This error is usually reported as a result of an event that is beyond control of your program such as a path or system is not available.

**Effect.** Program behavior is language and application dependent.

**Recovery.** Consult your system administrator.

## 83

Operation incompatible with file type or status (*error*)

**Cause.** The program attempted an operation on a file whose type or current status is unsuitable for execution of that operation. For example, a COBOL program calls a file manipulation utility for a file that is using Fast I/O. (*error*), if present, is a file-system error number.

**Effect.** Program behavior is language and application dependent.

**Recovery.** Change the program so that it does not attempt the operation on an unsuitable file.

## 90

Open conflicts with open by other language

**Cause.** Routines written in two different languages—for example, COBOL and FORTRAN—attempted to open a connection to a file using the same logical name. This error is not reported for standard files.

**Effect.** Program behavior is language and application dependent.

**Recovery.** Modify your program to use different logical names or coordinate logical names between the two routines so that they do not open the same logical file at the same time.

## 91

Spooler job already started

**Cause.** A FORTRAN routine attempted to open a spooler but the spooler was already open with attributes that conflict with those in the current open. This error is reported only for an open to standard output and only if one or more of the following are true:

- The spooling levels of the two opens are different.
- The new open specifies any level-2 arguments.

**Effect.** If the rejected request was initiated from a FORTRAN I/O statement that includes either an IOSTAT or ERR parameter, control is returned to the FORTRAN routine. Otherwise, the FORTRAN run-time library terminates the program.

**Recovery.** Coordinate how routines in your program use standard output.

# COBOL Messages

See the appropriate manual for the compiler for a description of the messages produced and for more information on CRE error handling for COBOL:

- *COBOL Manual for TNS and TNS/R Programs*
- *COBOL Manual for TNS/E Programs*

# FORTRAN Messages

The FORTRAN run-time library reports messages if an error occurs while a FORTRAN module is running. The FORTRAN run-time library terminates the program unless the error occurred on certain statements that specify the ERR parameter and or a status parameter (for example, IOSTAT or BACKUPSTATUS). Refer to the *FORTRAN Reference Manual* for additional error-handling information.

# Native CRE Messages

## 275

Ambiguity in application of `errno`

**Cause.** `errno` is defined in the user's object file. The native CRE reports this error if `errno` has been defined in the object file, and it is not the same `errno` defined by the native CRE shared run-time library instance data item `errno`.

**Effect.** The run-time library terminates your program.

**Recovery.** Make sure the only defined `errno` in a program is the one defined in the native CRE shared run-time library.

## 276

Ambiguity in application of `environ`

**Cause.** `environ` is defined in the user's object file. The native CRE reports this error if `environ` has been defined in the object file, and it is not the same `environ` defined by the native CRE shared run-time library instance data item `environ`.

**Effect.** The run-time library terminates your program.

**Recovery.** Make sure the only defined `environ` in a program is the one defined in the native CRE shared run-time library.

# Mapping Message Numbers Between Run-Time Environments

The tables in this subsection show the correspondence between the message numbers written to standard log when a program runs in language-specific run-time environment to when a program runs in the CRE. The text in the column titled “Original Message Text” shows the message text emitted by the original language-specific run-time libraries.

**Table 10-1. C Message Mapping**

C Message Number	CRE message Number	Original Message Text
1	31	insufficient heap space for argv
2	31	insufficient heap space for env
3	31	insufficient heap space for assigns
4	31	insufficient user heap space
5	31	internal memory allocation error <i>number</i>
6	59	unable to open standard input, errno = <i>number</i>
7	60	unable to open standard output, errno = <i>value</i>
8	61	unable to open standard error, errno = <i>value</i>
34		out of param space
	n/a	large-model run-time heap redefinition not supported
	n/a	illegal HEAP PARAM is ignored
	n/a	run-time heap redefined using HEAP PARAM
	n/a	**** fatal C library internal heap error****
		**** fatal user heap error ****
	11	run-time data has been overwritten
	12	run-time library internal consistency error

**Table 10-2. COBOL85 Message Mapping (page 1 of 6)**

COBOL85 Message Number	CRE Message Number	Original Message Text
001	125	OCCURS DEPENDING ON data item out of range
002	126	Extended address parameter passed where byte or word address expected
003	127	CALL references an active program
004	128	Reference modifier out of range



**Table 10-2. COBOL85 Message Mapping** (page 2 of 6)

<b>COBOL85 Message Number</b>	<b>CRE Message Number</b>	<b>Original Message Text</b>
005	129	Improper context for RELEASE statement
006	130	Improper context for RETURN statement
007	131	S location at end of paragraph not equal to that at beginning
008	132	SORT or MERGE statement executed while SORT or MERGE active
009	133	Subscript out of range
010	134	GO TO statement not initialized
011	135	ACCEPT or DISPLAY requested for an unsupported device
012	17	Space allocation for ACCEPT or DISPLAY failed
013	136	Input-output error <i>nnn</i> on ACCEPT or DISPLAY device
014	25	File not uniquely identified in ASSIGN command
015	26	PARAM argument not recognized or not permitted
016	137	Called program not found
017	138	CANCEL references an active program
018	139	Cancelled program not found
019	142	RELEASE operation failed - SORTMERGE message follows
020	143	RETURN operation failed - SORTMERGE message follows
021	144	SORT or MERGE operation failed before end - SORTMERGE message follows
022	145	SORT or MERGE operation failed at start - SORTMERGE
023	29	Run unit not compatible with COBOL85 library
024	150	Alternate key not present in file
025	151	DUPLICATES specification in SELECT does not match file
026	152	OPEN on a non-disk file that is specified with alternate
027	153	Create of new file failed with error <i>nnn</i>
028	154	Sequential DELETE must follow successful READ
029	155	DELETE positioning failed with error <i>nnn</i>
030	156	DELETE repositioning failed with error <i>nnn</i>
031	157	Wrong open mode for DELETE
032	158	DELETE operation failed with error <i>nnn</i>
033	77	OPEN on an EDIT file and EDITREADINIT failed with code <i>-n</i>
034	159	OPEN on an EDIT file and wrong open mode
035	160	OPEN on an EDIT file described with a record size that is too big

**Table 10-2. COBOL85 Message Mapping** (page 3 of 6)

<b>COBOL85 Message Number</b>	<b>CRE Message Number</b>	<b>Original Message Text</b>
036	161	OPEN EXTEND positioning failed with error <i>nnn</i>
037	162	OPEN on a nonexistent file that is not OPTIONAL
038	163	OPEN page eject failed with error <i>nnn</i>
039	164	OPEN rewind failed with error <i>nnn</i>
040	165	OPEN requested for an unsupported device other illegal device
041	166	OPEN requested for a locked file
042	167	OPEN requested for an open file
043	168	LINAGE file not on printer or process
044	169	LOCK or UNLOCK operation failed with error <i>nnn</i>
045	170	MULTIPLE FILE TAPE file not on tape
046	171	OPEN positioning for MULTIPLE FILE TAPE failed
047	172	OPEN on a nonexistent file and alternate keys specified
048	75	Buffer space allocation failed
049	173	Indexed file not defined as ORGANIZATION INDEXED
050	174	OPEN INPUT when file not on input device
051	175	Operation other than OPEN on file that is not open
052	176	File is not opened for timed I/O
053	177	OPEN OUTPUT when file not on output device
054	178	Relative file not defined as ORGANIZATION RELATIVE
055	179	Sequential file not defined as ORGANIZATION SEQUENTIAL
056	180	Non-disk or unstructured file and not sequential organization
057	181	OPEN operation failed with error <i>nnn</i>
058	182	Primary key offset in program does not match file
059	183	Primary key size in program does not match file
060	184	Purge of file during OPEN failed with error <i>nnn</i>
061	185	Purge data from file during OPEN failed with error <i>nnn</i>
062	78	EDITREAD failed with code <i>-n</i>
063	186	READ operation failed with error <i>nnn</i>
064	187	READ WITH LOCK on file with read ahead
065	188	READ positioning failed with error <i>nnn</i>
066	189	Sequential READ requested when current position is undefined
067	190	Wrong open mode for READ
068	191	Reel swap failed with error <i>nnn</i>

**Table 10-2. COBOL85 Message Mapping** (page 4 of 6)

<b>COBOL85 Message Number</b>	<b>CRE Message Number</b>	<b>Original Message Text</b>
069	192	Sequential REWRITE must follow successful READ
070	193	REWRITE positioning failed with error <i>nnn</i>
071	194	REWRITE repositioning failed with error <i>nnn</i>
072	195	Wrong open mode for REWRITE
073	196	Sequential REWRITE permitted only with same record size
074	197	REWRITE operation failed with error <i>nnn</i>
075	n/a	CLOSE LOCK while same file open elsewhere in run unit
076	198	START operation failed with error <i>nnn</i>
077	199	START positioning failed with error <i>nnn</i>
078	200	Wrong open mode for START
079	201	System node not available or does not exist
080	202	LOCKFILE, UNLOCKFILE or UNLOCKRECORD on file that is not open
081	203	OPEN on unstructured file described without fixed length records
082	204	Writing end of file failed with error <i>nnn</i>
083	205	Writing end of reel failed with error <i>nnn</i>
084	206	WRITE operation failed with error <i>nnn</i>
085	207	WRITE failed because file full
086	208	WRITE positioning failed with error <i>nnn</i>
087	209	WRITE repositioning failed with error <i>nnn</i>
088	210	Line skipping failed with error <i>nnn</i>
089	211	Wrong open mode for WRITE
090	212	Wrong length record specified for WRITE or REWRITE
091	n/a	Descriptions for the same external file differ
092	213	OPEN EXTEND for file not on extend device
093	214	OPEN I-O for file not on input-output device
094	215	Wrong or missing LABELS attribute
095	216	Wrong or missing USE attribute
096	217	Wrong or missing RECFORM attribute
097	218	Wrong or missing RECLLEN attribute
098	219	Wrong or missing BLOCKLEN attribute
099	220	Wrong or missing FILESEQ attribute

**Table 10-2. COBOL85 Message Mapping** (page 5 of 6)

<b>COBOL85 Message Number</b>	<b>CRE Message Number</b>	<b>Original Message Text</b>
100	221	Wrong or missing DEVICE attribute
101	222	A DEFINE procedure failed with error <i>nnn</i>
102	223	DEFINE required for LABEL RECORDS STANDARD
103	n/a	Blocking not permitted for odd length records
104	80	Spooler initialization failed with error <i>nnn</i>
105	n/a	No space for PARAM EXECUTION-LOG file block
106	26	External file name in PARAM EXECUTION-LOG is invalid
107	n/a	File referenced in PARAM EXECUTION-LOG is on an invalid device
108	140	Record referenced in RELEASE statement not in SD in SORT or MERGE statement
109	141	File referenced in RETURN statement not in SD in SORT or MERGE statement
110	79	OpenEdit failed with error <i>nnn</i>
111	224	PositionEdit failed with error <i>nnn</i>
112	225	Size of unstructured file opened EXTEND not multiple of record size
113	226	File attributes don't match and file not opened OUTPUT or has alt keys
114	227	Loadclose failed with internal error mmm, GUARDIAN error <i>nnn</i>
115	228	Allocatesegment for fast i/o failed with error return <i>nnn</i>
116	229	Loadopen failed with internal error mmm, GUARDIAN error <i>nnn</i>
117	230	Loadwrite failed with internal error mmm, GUARDIAN error <i>nnn</i>
118	231	Initnewdatablock (Fast I-O) failed
119	232	An illegal operation was attempted on a fast i/o file
120	n/a	The file-name in PARAM PRINTER-CONTROL is not a legal COBOL file-name
121	1 thru 9	A hardware/software trap occurred - the type follows
122	233	OPEN OUTPUT SHARED and other process has disk open
123	75	Buffer allocation failed on SORT or MERGE
124	234	The CLOSE operation failed with Guardian error <i>nnn</i>
125	n/a	An ENV COMMON routine called from an ENV OLD program
127	146	Parameter mismatch for CALL identifier

**Table 10-2. COBOL85 Message Mapping** (page 6 of 6)

<b>COBOL85 Message Number</b>	<b>CRE Message Number</b>	<b>Original Message Text</b>
128	83	Operation incompatible with the file type or status (error)
129	83	Operation incompatible with file type or status
130	40	Invalid function parameter
131	11 or 13	Corrupted environment

**Table 10-3. FORTRAN Message Mapping**

<b>FORTRAN Message Number</b>	<b>CRE message Number</b>	<b>Original Message Text</b>
256	256	BAD UNIT
257	257	BAD PARAMETER
267	267	BUFFER OVERFLOW
270	270	FORMAT LOOPBACK
271	271	EDIT ITEM MISMATCH
272	272	ILLEGAL INPUT CHARACTER
273	273	ILLEGAL FORMAT
274	274	NUMERIC OVERFLOW

\*The messages in this table appear in the *FORTRAN Reference Manual* but were not displayed by the FORTRAN run-time library. If you use the CRE with your FORTRAN program, the CRE message numbers and associated text are displayed.



# A

## Data Type Correspondence

The following tables contain the return value size generated by HP language compilers for each data type. Use this information when you need to specify values with the Accelerator ReturnValSize option. These tables are also useful if your programs use data from files created by programs in another language, or your programs pass parameters to programs written in callable languages.

Refer to the appropriate NonStop SQL/MP programmer's guide for a complete list of SQL data type correspondence. Also note that the return value sizes given in these tables do not correspond to the storage size of SQL data types.

**Note.** COBOL includes COBOL 74, COBOL85, and SCREEN COBOL unless otherwise noted.

If you are using the Data Definition Language (DDL) utility to describe your files, see the *Data Definition Language (DDL) Reference Manual* for more information.

**Table A-1. Integer Types, Part 1** (page 1 of 2)

Language	8-Bit Integer	16-Bit Integer	32-Bit Integer
BASIC	STRING	INT INT(16)	INT(32)
C	char <sup>1</sup> unsigned char signed char	int in the 16-bit data model short unsigned	int in the 32-bit or wide data model long unsigned long
COBOL	Alphabetic Numeric DISPLAY Alphanumeric-Edited Alphanumeric Numeric-Edited	PIC S9(n) COMP or PIC 9(n) COMP without P or V, $1 \leq n \leq 4$ Index Data Item <sup>2</sup> NATIVE-2 <sup>3</sup>	PIC S9(n) COMP or PIC 9(n) COMP without P or V, $5 \leq n \leq 9$ Index Data Item <sup>2</sup> NATIVE-4 <sup>3</sup>
FORTTRAN	--	INTEGER <sup>4</sup> INTEGER*2	INTEGER*4

1. Unsigned Integer.

2. Index Data Item is a 16-bit integer in COBOL 74 and a 32-bit integer in HP COBOL.

3. HP COBOL only.

4. INTEGER is normally equivalent to INTEGER\*2. The INTEGER\*4 and INTEGER\*8 compiler directives redefine INTEGER.

**Table A-1. Integer Types, Part 1** (page 2 of 2)

<b>Language</b>	<b>8-Bit Integer</b>	<b>16-Bit Integer</b>	<b>32-Bit Integer</b>
SQL	CHAR	NUMERIC(1)... NUMERIC(4)  PIC 9(1) COMP... PIC 9(4) COMP  SMALLINT	NUMERIC(5)... NUMERIC(9)  PIC 9(1) COMP ...PIC 9(9) COMP  INTEGER
TAL	STRING	INT	INT(32)
pTAL	UNSIGNED(8)	INT(16)  UNSIGNED(16)	
<b>Return Value Size (Words)</b>	<b>1</b>	<b>1</b>	<b>2</b>

1. Unsigned Integer.

2. Index Data Item is a 16-bit integer in COBOL 74 and a 32-bit integer in HP COBOL.

3. HP COBOL only.

4. INTEGER is normally equivalent to INTEGER\*2. The INTEGER\*4 and INTEGER\*8 compiler directives redefine INTEGER.



**Table A-2. Integer Types, Part 2**

<b>Language</b>	<b>64-Bit Integer</b>	<b>Bit Integer of 1 to 31 Bits</b>	<b>Decimal Integer</b>
BASIC	INT(64) FIXED(0)	--	--
C	long long	--	--
COBOL	PIC S9(n) COMP or PIC 9(n) COMP without P or V, 10 ð n ð 18 NATIVE-8 <sup>1</sup>	--	Numeric DISPLAY
FORTRAN	INTEGER*8	--	--
SQL	NUMERIC(10)... NUMERIC(18) PIC 9(10) COMP... PIC 9(18) COMP LARGEINT	--	DECIMAL (n,s) PIC 9(n) DISPLAY
TAL pTAL	FIXED(0), INT(64)	UNSIGNED(n), $1 \leq n \leq 31$	--
<b>Return Value Size (Words)</b>	<b>4</b>	<b>1, 1 or 2 in TAL</b>	<b>1 or 2, depends on declared pointer size</b>
1. HP COBOL only			

**Table A-3. Floating, Fixed, and Complex Types**

<b>Language</b>	<b>32-Bit Floating</b>	<b>64-Bit Floating</b>	<b>64-Bit Fixed Point</b>	<b>64-Bit Complex</b>
BASIC	REAL	REAL(64)	FIXED(s), $0 \leq s \leq 18$	--
C	float	double	--	--
COBOL	--	--	PIC S9(n-s)v9(s) COMP or PIC 9(n-s)v9(s) COMP, $10 \leq n \leq 18$	--
FORTRAN	REAL	DOUBLE PRECISION	--	COMPLEX
SQL	--	--	NUMERIC (n,s) PIC 9(n-s)v9(s) COMP	--
TAL	REAL	REAL(64)	FIXED(s), $-19 \leq s \leq 19$	--
pTAL	REAL(32)			
<b>Return Value Size (Words)</b>	<b>2</b>	<b>4</b>	<b>4</b>	<b>4</b>

**Table A-4. Character Types**

<b>Language</b>	<b>Character</b>	<b>Character String</b>	<b>Varying Length Character String</b>
BASIC	STRING	STRING	--
C	signed char unsigned char	pointer to char	struct { int len; char val [n] };
COBOL	Alphabetic Numeric DISPLAY Alphanumeric-Edited Alphanumeric Numeric-Edited	Alphabetic Numeric DISPLAY Alphanumeric-Edited Alphanumeric Numeric-Edited	01 name. 03 len USAGE IS NATIVE-2 <sup>1</sup> 03 val PIC X(n).
FORTRAN	CHARACTER	CHARACTER array CHARACTER*n	--
SQL	PIC X CHAR	CHAR(n) PIC X(n)	VARCHAR(n)
TAL pTAL	STRING	STRING array	--
<b>Return Value Size (Words)</b>	<b>1</b>	<b>1 or 2, depends on declared pointer size</b>	<b>1 or 2, depends on declared pointer size</b>
1. HP COBOL only.			

**Table A-5. Structured, Logical, Set, and File Types**

<b>Language</b>	<b>Byte-Addressed Structure</b>	<b>Word-Addressed Structure</b>	<b>Logical (true or false)</b>	<b>Boolean</b>	<b>Set</b>	<b>File</b>
BASIC	--	MAP buffer	--	--	--	--
C	--	struct	--	--	--	--
COBOL	--	01-level RECORD	--	--	--	--
FORTRAN	RECORD	--	LOGICAL <sup>1</sup>	--	--	--
SQL	--	--	--	--	--	--
TAL pTAL	Byte-addressed standard STRUCT pointer	Word-addressed standard STRUCT pointer	--	--	--	--
<b>Return Value Size (Words)</b>	<b>1 or 2, depends on declared pointer size</b>	<b>1 or 2, depends on declared pointer size</b>	<b>1 or 2, depends on compiler directive</b>	<b>1</b>	<b>1</b>	<b>1</b>
1. LOGICAL is normally defined as 2 bytes. The LOGICAL*2 and LOGICAL*4 compiler directives redefine LOGICAL.						

**Table A-6. Pointer Types**

<b>Language</b>	<b>Procedure Pointer</b>	<b>Byte Pointer</b>	<b>Word Pointer</b>	<b>Extended Pointer</b>
BASIC	--	--	--	--
C	function pointer	byte pointer	word pointer	extended pointer
COBOL	--	--	--	--
FORTRAN	--	--	--	--
SQL	--	--	--	--
TAL	--	16-bit pointer, byte- addressed	16-bit pointer, word-addressed	32-bit pointer
pTAL	PROCPTR	16-bit pointer, byte- addressed	16-bit pointer, word-addressed	32-bit pointer
<b>Return Value Size (Words)</b>	<b>1 or 2, depends on declared pointer size</b>	<b>1 or 2, depends on declared pointer size</b>	<b>1 or 2, depends on declared pointer size</b>	<b>1 or 2, depends on declared pointer size</b>



# Glossary

**accelerate.** To speed up emulated execution of a TNS object file by applying the Accelerator on TNS/R systems or the Object Code Accelerator (OCA) on TNS/E systems before running the object file.

**accelerated mode.** See [TNS accelerated mode](#).

**accelerated object code.** The MIPS RISC instructions (called the TNS/R region) that result from processing a TNS object file with the Accelerator or the Itanium instructions (called the TNS/E region) that result from processing a TNS object file with the Object Code Accelerator (OCA).

**accelerated object file.** A TNS object file that, in addition to its TNS instructions (in the TNS region) and symbol information (in the symbol region), has been augmented either by the Accelerator with equivalent but faster MIPS RISC instructions (in the TNS/R region) or the Object Code Accelerator (OCA) with equivalent but faster Itanium instructions (in the TNS/E region), or both.

**Accelerator.** A program optimization tool that processes a TNS object file and produces an accelerated object file that also contains equivalent MIPS RISC instructions (called the TNS/R region). TNS object code that is accelerated runs faster on TNS/R processors than TNS object code that is not accelerated. See also [TNS Object Code Accelerator \(OCA\)](#).

**Accelerator-generated MIPS RISC instructions.** See [MIPS RISC instructions](#).

**Accelerator-generated RISC instructions.** See [MIPS RISC instructions](#).

**Accelerator mode.** The TNS/R operational environment in which an object file containing MIPS RISC instructions (called the TNS/R region) executes. *Contrast with* [TNS/R native object code](#).

**application program interface (API).** A set of services (such as programming language functions or procedures) that are called by an application program to communicate with other software components. For example, under Open System Services, an application uses functions defined by ISO/IEC IS 9945-1:1990 to perform process management, time management, and file management functions through the operating system.

**ASSIGN command.** A TACL command that lets you associate a logical file name with a Guardian file name (physical file name). The Guardian file name is a fully qualified file ID. See also [file name](#) and [file ID](#).

**Binder.** A programming utility that combines one or more compilation units' TNS object code files to create an executable TNS object code file for a TNS program or library. Used only with [TNS object files](#).

**binding.** The operation of collecting, connecting, and relocating code and data blocks from one or more separately compiled TNS object files to produce a target object file.

**breakpoint.** An object code location at which execution will be suspended so that you can interactively examine and modify the process state. With symbolic debuggers, breakpoints are usually at source line or statement boundaries.

In TNS/R or TNS/E native object code, breakpoints can be at any MIPS RISC instruction or Itanium instruction within a statement. In a TNS object file that has not been accelerated, breakpoints can be at any TNS instruction location. In a TNS object file that has been accelerated, breakpoints can be only at certain TNS instruction locations not at arbitrary instructions; some source statement boundaries are not available. However, breakpoints can be placed at any instruction in the accelerated code.

**CLUDECS.** A file, provided by the TNS CRE, that contains external declarations for CLULIB functions.

**CLURDECS.** A file, provided by the native CRE, that contains external declarations for Saved Message Utility functions, data, and data structure declarations in pTAL.

**Common Language Utility (CLU) library.** A collection of functions that provide common services to two or more language products.

**Common Run-Time Environment (CRE).** A set of services implemented by the CRE library that supports mixed-language programs. Contrast with [language-specific run-time environment](#)

**Common Run-Time Environment (CRE) library.** A collection of functions that supports requests for services managed by the CRE, such as I/O and heap management, math and string functions, exception handling, and error reporting. C, COBOL, and FORTRAN run-time libraries call CRE library functions to access resources managed by the CRE. TAL and pTAL user routines can call CRE library functions to access resources managed by the CRE.

**HP NonStop operating system.** The operating system for HP NonStop servers.

**CISC.** See **complex instruction-set computing (CISC)**.

**complex instruction-set computing (CISC).** A processor architecture based on a large instruction set, characterized by numerous addressing modes, multicycle machine instructions, and many special-purpose instructions. Contrast with [reduced instruction-set computing \(RISC\)](#)

**connection.** A path managed by the CRE from a process to a Guardian file. Each connection is a unique path to the same Guardian file and to the same open of that file. The CRE manages the connection. The CRE provides connection services for standard files.



**CRE.** See [Common Run-Time Environment \(CRE\)](#)

**CREDECS.** A file, provided by the TNS CRE, that contains external declarations for CRELIB functions whose names begin with CRE\_.

**CRERDECS.** A file, provided by the native CRE, that contains external declarations for CRE function, data, and data structure declarations in pTAL.

**DEFINE command.** A TACL command that lets you specify a named set of attributes and values to pass to a process.

**eld utility.** A utility that collects, links, and modifies code and data blocks from one or more position-independent code (PIC) object files to produce a target TNS/E native object file. See also [Binder](#) and [nld utility](#).

**emulate.** To imitate the instruction set and address spaces of a different hardware system by means of software. Emulator software is compatible with and runs software built for the emulated system. For example, a TNS/R or TNS/E system emulates the behavior of a TNS system when executing interpreted or accelerated TNS object code.

**enoft utility.** A utility that reads and displays information from TNS/E native object files. See also [noft utility](#).

**execution mode.** The (emulated or real) instruction set environment in which object code runs. A TNS system has only one execution mode: TNS mode using TNS compilers and 16-bit TNS instructions. A TNS/R system has three execution modes: TNS/R native mode using MIPS native compilers and MIPS instructions, emulated TNS execution in TNS interpreted mode, and emulated TNS execution in TNS accelerated mode. A TNS/E system also has three execution modes: TNS/E native mode using Itanium native compilers and Itanium instructions, emulated TNS execution in TNS interpreted mode, and emulated TNS execution in TNS accelerated mode.

**extended data segment.** A segment that provides up to 127.5 megabytes of indirect data storage. A process can have more than one extended data segment.

**FCB.** See **file-control block (FCB)**.

**file connector.** An abstract entity through which a program accesses a file. It is physically represented by the file-control block (FCB).

**file-control block (FCB).** A run-time data object that contains information about an open file.

**file ID.** The last of the four parts of a file name; the first three parts are node name (system name), volume name, and subvolume name.

**file name.** A fully qualified file ID. A file name contains four parts separated by periods:

- Node name (system name)
- Volume name

- Subvolume name
- File ID

**global data.** Data declarations that appear before the first procedure declaration; identifiers that are accessible to all compilations units in a binding session.

**Guardian.** An environment or API available for interactive or programmatic use with the NonStop operating system. Processes that run in the Guardian environment use the Guardian system procedure calls as their application program interface; interactive users of the Guardian environment use the HP Tandem Advanced Command Language (TACL) or another HP product's command interpreter. Contrast with [Open System Services \(OSS\)](#)

**Guardian environment.** The Guardian API, tools, and utilities.

**home terminal.** Usually the terminal from which a process was started.

**Intel Itanium microprocessors.** The series of Intel processors that support the Itanium instruction set.

**interpreted mode.** See [TNS interpreted mode](#).

**Intel Itanium instructions.** Register-oriented machine instructions that are native to and directly executed by a TNS/E system. Intel Itanium instructions do not execute on TNS and TNS/R systems. *Contrast with* [TNS instructions](#) *and* [MIPS RISC instructions](#).

The TNS Object Code Accelerator (OCA) produces Intel Itanium instructions to accelerate TNS object code. A TNS/E native compiler produces native-compiled Intel Itanium instructions when it compiles source code.

**Itanium word.** An instruction-set-defined unit of memory. An Itanium word is 4 bytes (32 bits) wide, beginning on any 4-byte boundary in memory. *Contrast with* [TNS word and word](#). See also [MIPS RISC word](#).

**language-specific run-time environment.** A set of services implemented by the run-time library of each language. Without the CRE, C, COBOL, FORTRAN, or TAL programs each have their own language-specific run-time environments. These language-specific run-time environments are often incompatible with each other. Contrast with [Common Run-Time Environment \(CRE\)](#)

**language-specific run-time library.** A collection of functions outside the CRE that supports requests from a specific language for services such as I/O and heap management, math and string functions, exception handling, and error reporting.

**large memory model.** A program attribute that specifies that a program's heap is allocated in the extended memory segment.

**linking.** The operating of examining, collecting, linking, and modifying code and data blocks from one or more object files to produce a target object file.

**local data.** Data that you declare within a procedure; identifiers that are accessible only from within that procedure.

**lower 32K-word area.** The lower half of the user data segment. The global, local, and sublocal storage areas.

**main routine.** The first routine to execute when a program is run. The main routine determines the run-time environment for a program. It is the routine declared with the MAIN or PROGRAM keyword.

**Master Control Block (MCB).** A structure that holds CRE data.

**MCB.** See **Master Control Block (MCB).**

**mixed-language program.** A program that contains source files written in different HP programming languages.

**MIPS RISC instructions.** Register-oriented 32-bit machine instructions in the MIPS-1 RISC instruction set that are native to and directly executed on TNS/R systems. MIPS RISC instructions do not execute on TNS systems and TNS/E systems. *Contrast with* [TNS instructions](#) *and* [Intel Itanium instructions](#).

Accelerator-generated MIPS RISC instructions are produced by accelerating TNS object code. Native-compiled MIPS RISC instructions are produced by compiling source code with a TNS/R native compiler.

**MIPS RISC word.** An instruction-set-defined unit of memory. A MIPS RISC word is 4 bytes (32 bits) wide, beginning on any 4-byte boundary in memory. *Contrast with* [TNS word](#) *and* [word](#). See also [Itanium word](#).

**native.** An adjective that can modify the following: object code, object file, process, procedure, and mode of process execution. Native object files contain native object code, which directly uses the MIPS instruction set or Intel Itanium processor instructions and the corresponding conventions for register handling and procedure calls. Native processes are those created by executing native object files. Native procedures are units of native object code. Native-mode execution is the state of the process when it is executing native procedures.

**native-compiled Itanium instructions.** See [Intel Itanium instructions](#).

**native-compiled MIPS RISC instructions.** See [MIPS RISC instructions](#).

**native-compiled RISC instructions.** See [MIPS RISC instructions](#).

**native mode.** See [TNS/R native mode](#) or [TNS/E native mode](#).

**native-mode code.** Object code that has been compiled with TNS/R native compilers to run on TNS/R systems or with TNS/E native compilers to run on TNS/E systems.

**native-mode library.** A native-compiled loadfile associated with one or more other native-compiled loadfiles. A native-mode process can have any number of associated native-mode libraries. See also [TNS user library](#), [TNS/R native user library](#), and [TNS/E library](#).

**native-mode source code.** High-level language routines that can be compiled with either TNS/R native compilers or TNS/E native compilers. These two sets of compilers accept the same language dialects.

**native object code.** See [TNS/R native object code](#) or [TNS/E native object code](#).

**native object file.** See [TNS/R native object file](#) or [TNS/E native object file](#).

**native object file tool.** See [noft utility](#) and [enoft utility](#).

**native process.** See [TNS/R native process](#) or [TNS/E native process](#).

**native signal.** See [TNS/R native signal](#) or [TNS/E native signal](#).

**nld utility.** A utility that collects, links, and modifies code and data blocks from one or more object files to produce a target TNS/R native object file. See also [Binder](#) and [eld utility](#).

**noft utility.** A utility that reads and displays information from TNS/R native object files. See also [enoft utility](#).

**NonStop Kernel.** See [HP NonStop operating system](#)

**NonStop Open System Services (OSS).** An application program interface (API) to the HP NonStop operating system and associated tools and utilities. For a more complete definition, see [Open System Services \(OSS\)](#)

**object code accelerator (OCA).** See [TNS Object Code Accelerator \(OCA\)](#).

**object code interpreter (OCI).** See [TNS Object Code Interpreter \(OCI\)](#).

**object file.** A file generated by a compiler or binder that contains machine instructions and other information needed to construct the executable code spaces and initial data for a process. The file may be a complete program that is ready for immediate execution, or it may be incomplete and require binding with other object files before execution.

**OCA.** (1) The command used to invoke the TNS Object Code Accelerator (OCA) on a TNS/E system.

(2) See [TNS Object Code Accelerator \(OCA\)](#).

**OCA-accelerated object code.** The Itanium instructions that result from processing a TNS object file with the TNS Object Code Accelerator (OCA).

**OCA-accelerated object file.** A TNS object file that has been augmented by the TNS Object Code Accelerator (OCA) with equivalent but faster Itanium instructions. An OCA-accelerated object file contains the original TNS object code, the OCA-accelerated object code and related address map tables, and any Binder and symbol information from the original TNS object file. An OCA-accelerated object file also can be augmented by the Accelerator with equivalent MIPS RISC instructions.

**open.** A path from a process to a Guardian file that is managed by the file system. Each open is a unique path created by calling the FILE\_OPEN\_ system procedure. See also **connection**.

**Open System Services (OSS).** An open system environment available for interactive or programmatic use with the HP NonStop operating system. Processes that run in the OSS environment use the OSS application program interface; interactive users of the OSS environment use the OSS shell for their command interpreter. Contrast with [Guardian](#)

**OSS.** See [Open System Services \(OSS\)](#)

**OSS environment.** The Open System Services (OSS) API, tools, and utilities.

**OSS signal.** A signal model defined in the POSIX.1 specification and available to TNS processes and native processes in the OSS environment. OSS signals can be sent between processes.

**OSS environment.** The NonStop Open System Services (OSS) API, tools, and utilities. Referred to as “personality” in marketing literature.

**PARAM command.** A TACL command that lets you associate an ASCII value with a parameter name.

**process.** A program that has been submitted to the operating system for execution. An instance of execution of a program.

**process snapshot.** The contents of a saveabend file or process snapshot file.

**process snapshot file.** (1) A file containing dump information needed by the system debugging tool. In UNIX systems, such files are usually called core files or core dump files.

(2) A file containing the state of a running process at a specific moment, regardless of whether the process terminated abnormally.

See also [saveabend file](#).

**program file.** An executable object file. See [open](#)

**pTAL.** Portable Transaction Application Language. A machine-independent system programming language based on Transaction Application Language (TAL). The pTAL

language excludes architecture-specific TAL constructs and includes new constructs that replace the architecture-specific constructs. *Contrast with* [TAL](#).

**pTAL compiler.** An optimizing native-mode compiler for the pTAL language.

**public dynamic-link library (public DLL).** Optional native-mode executable code modules available to all native user processes. A TNS/E public library that is specified in the public library registry, supplied by HP or, optionally, a user.

**reduced instruction-set computing (RISC).** A processor architecture based on a relatively small and simple instruction set, a large number of general-purpose registers, and an optimized instruction pipeline that supports high-performance instruction execution. Contrast with [complex instruction-set computing \(CISC\)](#).

**RISC.** See [reduced instruction-set computing \(RISC\)](#).

**RISC instructions.** MIPS RISC instructions. See also [MIPS RISC instructions](#).

**RISC word.** An instruction-set-defined unit of memory. A RISC word is a MIPS RISC word. *Contrast with* [TNS word](#) and [word](#). See also [Itanium word](#) and [MIPS RISC word](#).

**RTL.** See **run-time library (RTL)**.

**RTLDECS.** A file, provided by the TNS CRE, that contains external declarations for CRELIB functions whose names begin with RTL\_.

**RTLRECS.** A file, provided by the native CRE, that contains external declarations for RTL functions, data, and data structure declarations in pTAL.

**run-time environment.** The services provided by run-time library functions and data objects (data blocks and pointers) to a program at run-time.

**run-time library (RTL).** A collection of functions that supports requests for services such as I/O and heap management, math and string functions, exception handling, and error reporting.

**saveabend file.** A file containing dump information needed by the system debugging tool. (In UNIX systems, such files are usually called core files or core dump files.) A saveabend file is a special case of a save file. See also [save file](#) and [process snapshot file](#).

**save file.** A file created by the system in response to a command from a debugger. A save file contains enough information about a running process at a given time to restart the process at the same point in its execution. A save file contains an image of the process, data for the process, and the status of the process at the time the save file was created.

A save file can be created through an Inspect SAVE command at any time. A save file called a saveabend file can be created by when a process's SAVEABEND attribute is

set and the process terminates abnormally. Other debuggers can create a save file but refer to the result as a process snapshot file. See *also* [process snapshot file](#).

**Saved Message Utility.** See [SMU functions](#)

**shared file.** A standard file that your program can open multiple times. Each time your program opens the file, it is granted a connection to the same Guardian file open. Shared files are managed by the CRE. The CRE supports three shared files: standard input, standard output, and standard log.

**shared run-time library (SRL).** An object file that the operating system links to a program file at run time. See *also* [Transaction Application Language \(TAL\)](#) and [TNS/R native shared run-time library \(TNS/R native SRL\)](#)

**signal.** The method by which an environment notifies a process of an event. Signals are used to notify a process when an error that is not related to input or output has occurred. See *also* [OSS signal](#), [TNS State Library for TNS/E](#), [TNS/E native signal](#), and [TNS/R native signal](#).

**signal handler.** A procedure that is executed when a signal is received by a process.

**single-language program.** program in which all routines are written in the same programming language.

**small memory model.** A program attribute that specifies that the program's heap is allocated in the user data segment.

**SMU functions.** Saved Message Utility (SMU) functions provided by the Common Language Utility Library of the CRE. COBOL, FORTRAN, and TAL routines can call SMU functions to manipulate saved startup, ASSIGN, and PARAM messages.

**SRL.** See [shared run-time library \(SRL\)](#)

**standard file.** A file that your program can use with little or no need to establish file parameters. The CRE supports three standard files—standard input, standard output, and standard log—that correspond to the files STDIN, STDOUT, and STDERR in a C programming environment.

**standard input.** A file from which a program can read sequential records. Each program defines how standard input is used according to the needs of the application. Standard input is analogous to the file STDIN in C.

**standard log.** A file to which a program can write sequential records. The records written to standard log are usually informational, warning, or error messages that describe exceptional conditions in a program. Standard log is analogous to the file STDERR in C.



**standard output.** A file to which a program can write sequential records. The program defines how standard output is used according to the needs of the application. Standard output is analogous to the file STDOUT in C.

**sublocal data.** Data that you declare within a subprocedure; identifiers that are accessible only from within that subprocedure.

**system procedure.** A procedure supplied as part of the operating system.

**TAL.** See [Transaction Application Language \(TAL\)](#)

**TNS.** Denotes fault-tolerant HP computers that:

- Support the NonStop operating system
- Are based on microcoded complex instruction-set computing (CISC) technology.

TNS systems run the TNS instruction set. *Contrast with* [TNS/R](#) and [TNS/E](#).

**TNS accelerated mode.** A TNS emulation environment on a TNS/R or TNS/E system in which accelerated TNS object files are run. TNS instructions have been previously translated into optimized sequences of MIPS or Itanium instructions. TNS accelerated mode runs much faster than TNS interpreted mode. Accelerated or interpreted TNS object code cannot be mixed with or called by native-mode object code. *See also* [TNS Object Code Accelerator \(OCA\)](#). *Contrast with* [TNS/R native mode](#) and [TNS/E native mode](#).

**TNS C compiler.** The C compiler that generates TNS object files. *Contrast with* [TNS/R native C compiler](#) and [TNS/E native C compiler](#).

**TNS code segment.** One of up to 32 128-kilobyte areas of TNS object code within a TNS code space. Each segment contains the TNS instructions for up to 510 complete routines. Each TNS code segment contains its own procedure entry point (PEP) and external procedure entry point (XEP) tables. It may also contain read-only data.

**TNS code segment identifier.** A seven-bit value in which the most significant two bits encode a code space (user code, user library, system code, or system library) and the five remaining bits encode a code segment index in the range 0 through 31.

**TNS compiler.** A compiler in the TNS development environment that generates 16-bit TNS object code following the TNS conventions for memory, stacks, 16-bit registers, and call linkage. The TNS C compiler is an example of such a compiler. *Contrast with* [TNS/R native compiler](#) and [TNS/E native compiler](#).

**TNS Emulation Library.** A public dynamic-link library (DLL) on a TNS/E system containing the TNS Object Code Interpreter (OCI), millicode routines used only by accelerated mode, and millicode for switching among interpreted, accelerated, and native execution modes.

**TNS emulation software.** The set of tools, libraries, and system services for running TNS object code on TNS/E systems and TNS/R systems. On a TNS/E system, the TNS



emulation software includes the TNS Object Code Interpreter (OCI), the TNS Object Code Accelerator (OCA), and various millicode libraries. On a TNS/R system, the TNS emulation software includes the TNS Object Code Interpreter (OCI), the Accelerator, and various millicode libraries.

**TNS fixup.** A task performed at process startup time when executing a TNS object file. This task involves building the procedure entry point (PEP) table and external entry point (XEP) table and patching PCAL and XCAL instructions in a TNS object file before loading the file into memory.

**TNS instructions.** Stack-oriented, 16-bit machine instructions that are directly executed on TNS systems by hardware and microcode. TNS instructions can be emulated on TNS/E and TNS/R systems by using millicode, an interpreter, and either translation or acceleration. *Contrast with* [MIPS RISC instructions](#) and [Intel Itanium instructions](#).

**TNS interpreted mode.** A TNS emulation environment on a TNS/R or TNS/E system in which individual TNS instructions in a TNS object file are directly executed by interpretation rather than permanently translated into MIPS or Itanium instructions. TNS interpreted mode runs slower than TNS accelerated mode. Each TNS instruction is decoded each time it is executed, and no optimizations between TNS instructions are possible. TNS interpreted mode is used when a TNS object file has not been accelerated for that hardware system, and it is also sometimes used for brief periods within accelerated object files. Accelerated or interpreted TNS object code cannot be mixed with or called by native-mode object code. See also [TNS Object Code Interpreter \(OCI\)](#). Contrast with [TNS accelerated mode](#), [TNS/R native mode](#), and [TNS/E native mode](#).

**TNS library.** A single, optional, TNS-compiled loadfile associated with one or more application loadfiles. If a user library has its own global or static variables, it is called a [TNS shared run-time library \(TNS SRL\)](#). Otherwise it is called a [user library](#).

**TNS mode.** Synonym for TNS interpreted mode.

**TNS object code.** The TNS instructions that result from processing program source code with a TNS compiler. TNS object code executes on TNS, TNS/E, and TNS/R systems.

**TNS Object Code Accelerator (OCA).** A program optimization tool that processes a TNS object file and produces an accelerated file for a TNS/E system. OCA augments a TNS object file with equivalent Itanium instructions. TNS object code that is accelerated runs faster on TNS/E systems than TNS object code that is not accelerated. See also [Accelerator](#) and [TNS Object Code Interpreter \(OCI\)](#).

**TNS Object Code Interpreter (OCI).** A program that processes a TNS object file and emulates TNS instructions on a TNS/E system without preprocessing the object file. See also [TNS Object Code Accelerator \(OCA\)](#).

**TNS object file.** An object file created by a TNS compiler or the Binder. A TNS object file contains TNS instructions. TNS object files can be processed by the Accelerator or by

the TNS Object Code Accelerator (OCA) to produce to produce accelerated object files. A TNS object file can be run on TNS, TNS/R, and TNS/E systems.

**TNS procedure label.** A 16-bit identifier for an internal or external procedure used by the TNS object code of a TNS process. The most-significant 7 bits are a TNS code segment identifier: 2 bits for the TNS code space and 5 bits for the TNS code segment index. The least-significant 9 bits are an index into the target segment's procedure entry-point (PEP) table. On a TNS/E system, all shells for calling native library procedures are segregated within the system code space. When the TNS code space bits of a TNS procedure label are %B10, the remaining 14 bits are an index into the system's shell map table, not a segment index and PEP index.

**TNS process.** A process whose main program object file is a TNS object file, compiled using a TNS compiler. A TNS process executes in interpreted or accelerated mode while within itself, when calling a user library, or when calling into TNS system libraries. A TNS process temporarily executes in native mode when calling into native-compiled parts of the system library. Object files within a TNS process might be accelerated or not, with automatic switching between accelerated and interpreted modes on calls and returns between those parts.

*Contrast with [TNS/R native process](#) and [TNS/E native process](#).*

**TNS shared run-time library (TNS SRL).** An SRL available to TNS processes in the OSS environment on G-series systems. A TNS process can have only one TNS SRL. A TNS SRL is implemented as a special user library that allows shared global data.

**TNS signal.** A signal model available to TNS processes in the Guardian environment.

**TNS user library.** A user library available to TNS processes in the Guardian environment.

**TNS State Library for TNS/E.** A library of routines to access and modify the TNS state of a TNS process running on TNS/E.

**TNS word.** An instruction-set-defined unit of memory. A TNS word is 2 bytes (16 bits) wide, beginning on any 2-byte boundary in memory. See also [Itanium word](#), [MIPS RISC word](#), and [word](#).

**TNS/E.** Denotes fault-tolerant HP computers that support the HP NonStop operating system and that are based on the Intel Itanium processor-based architecture. TNS/E systems run the Itanium instruction set and can run TNS object files by interpretation or after acceleration. TNS/E systems include all HP systems that use NSAL-*x* processors.  
*Contrast with [TNS](#) and [TNS/R](#).*

**TNS/E library.** A TNS/E native-mode library. TNS/E libraries are always dynamic-link libraries (DLLs); there is no native shared runtime library (SRL) format.

**TNS/E native C compiler.** The C compiler that generates TNS/E object files. *Contrast with [TNS C compiler](#) and [TNS/R native C compiler](#).*

**TNS/E native compiler.** A compiler in the TNS/E development environment that generates TNS/E native object code, following the TNS/E native-mode conventions for memory, stack, registers, and call linkage. The TNS/E native C compiler is an example of such a compiler. *Contrast with [TNS compiler](#) and [TNS/R native compiler](#).*

**TNS/E native mode.** The primary execution environment on a TNS/E system, in which native-compiled Itanium object code executes, following TNS/E native-mode compiler conventions for data locations, addressing, stack frames, registers, and call linkage. *Contrast with [TNS interpreted mode](#) and [TNS accelerated mode](#). See also [TNS/R native mode](#).*

**TNS/E native object code.** The Itanium instructions that result from processing program source code with a TNS/E native compiler. TNS/E native object code executes only on TNS/E systems, not on TNS systems or TNS/R systems.

**TNS/E native object file.** An object file created by a TNS/E native compiler that contains Itanium instructions and other information needed to construct the code spaces and the initial data for a TNS/E native process.

**TNS/E native process.** A process initiated by executing a TNS/E native object file. *Contrast with [TNS process](#) and [TNS/R native process](#).*

**TNS/E native signal.** A signal model available to TNS/E native processes in both the Guardian and Open System Services (OSS) environments. TNS/E native signals are used for error exception handling.

**TNS/E native user library.** A user library available to TNS/E native processes in both the Guardian and Open System Services (OSS) environments. A TNS/E native user library is implemented as a TNS/E native dynamic-link library (DLL).

**TNS/R.** Denotes fault-tolerant HP computers that:

- Support the NonStop operating system
- Are based on 32-bit reduced instruction-set computing (RISC) technology.

TNS/R systems run the MIPS-1 RISC instruction set and can run TNS object files by interpretation or after acceleration. TNS/R systems include all HP systems that use NSR-*x* processors. *Contrast with [TNS](#) and [TNS/E](#).*

**TNS/R native C compiler.** The C compiler that generates TNS/R object files. *Contrast with [TNS C compiler](#) and [TNS/E native C compiler](#).*

**TNS/R native compiler.** A compiler in the TNS/R development environment that generates TNS/R native object code, following the TNS/R native-mode conventions for memory, stack, 32-bit registers, and call linkage. The TNS/R native C compiler is an example of such a compiler. *Contrast with [TNS compiler](#) and [TNS/E native compiler](#).*

**TNS/R native mode.** The primary execution environment on a TNS/R system, in which native-compiled MIPS object code executes, following TNS/R native-mode compiler conventions for data locations, addressing, stack frames, registers, and call linkage.

Contrast with [TNS interpreted mode](#) and [TNS accelerated mode](#). See also [TNS/E native mode](#).

**TNS/R native object code.** The MIPS RISC instructions that result from processing program source code with a TNS/R native compiler. TNS/R native object code executes only on TNS/R systems, not on TNS systems or TNS/E systems.

**TNS/R native object file.** An object file created by a TNS/R native compiler that contains MIPS RISC instructions and other information needed to construct the code spaces and the initial data for a TNS/R native process.

**TNS/R native process.** A process initiated by executing a TNS/R native object file. Contrast with [TNS process](#) and [TNS/E native process](#).

**TNS/R native signal.** A signal model available to TNS/R native processes in both the Guardian and Open System Services (OSS) environments. TNS/R native signals are used for error exception handling.

**TNS/R native user library.** A user library available to TNS/R native processes in both the Guardian and Open System Services (OSS) environments. A TNS/R native user library is implemented as a special private TNS/R native shared run-time library (TNS/R native SRL).

**TNS/R native shared run-time library (TNS/R native SRL).** A shared run-time library (SRL) available to TNS/R native processes in both the Guardian and Open System Services (OSS) environments. TNS/R native SRLs can be either public or private. A TNS/R native process can have multiple public SRLs but only one private SRL.

**Transaction Application Language (TAL).** A systems programming language with many features specific to stack-oriented TNS systems.

**upper 32K-word area.** The upper half of the user data segment. TAL routines can use pointers to allocate this area for your data; however, if you use the CRE, the upper 32K-word area is not available for your data.

**user data segment.** An automatically allocated segment that provides modifiable, private storage for the variables of your process.

**user library.** (1) An object code file that the operating system links to a program file at run time. A program can have only one user library. See also [TNS user library](#), [TNS/R native user library](#), and [TNS/E native user library](#).

(2) A library loadfile associated with a program so that it emulates the user library feature of the operating system on TNS systems. For position-independent code programs on TNS/R and TNS/E systems, the user library is a dynamic-link library. It is treated as if it were the first library in the program's libList; thus it is searched first for symbols required by the program. However, a user library does not appear in the program's libList; instead, its name is recorded internally in the program's loadfile. A program can be associated with at most one user library; the association can be

specified using the linker at link time or in a later change command, or at run time using the process creation interfaces. (The /LIB .../ option to the RUN command in TACL uses these interfaces.)

**word.** An instruction-set-defined unit of memory that corresponds to the width of registers and to the most common and efficient size of memory operations. A TNS word is 2 bytes (16 bits) wide, beginning on any 2-byte boundary in memory. A MIPS RISC word is 4 bytes (32 bits) wide, beginning on any 4-byte boundary in memory. An Itanium word is also 4 bytes (32 bits) wide, beginning on any 4-byte boundary in memory.



# Index

[A](#) | [B](#) | [C](#) | [D](#) | [E](#) | [F](#) | [G](#) | [H](#) | [I](#) | [L](#) | [M](#) | [N](#) | [O](#) | [P](#) | [R](#) | [S](#) | [T](#) | [U](#) | [V](#) | [W](#) | [Z](#)  
[Special Characters](#)

## A

Advantages of the CRE [1-8](#)  
Arccos [7-4](#)  
Arcsin [7-5](#)  
Arctan [7-5](#)  
Arctan2 [7-6](#)  
Arithmetic overflow [7-1](#)  
ARMTRAP [2-51](#)  
ASCII to numeric function  
    Atof [8-3](#)  
    Atoi [8-3](#)  
    Atol [8-3](#)  
ASSIGN message  
    changing [9-23](#), [9-25](#)  
    creating [9-23](#), [9-25](#)  
    deleting [9-19](#)  
    finding greatest message number [9-26](#)  
    parts [5-6](#)  
    retrieving [9-21](#), [9-22](#)  
Atof [8-3](#)  
Atoi [8-3](#)  
Atol [8-3](#)

## B

Binder  
    INFO command [3-7](#)  
    language-consistency checking [3-9](#)  
    parameter checking [3-8](#)  
    return-value checking [3-8](#)  
Binding  
    CLUL functions [5-2](#)  
    CRELIB [3-8](#)  
    examples [3-8](#)  
    for minimal size [3-8](#)

Binding (continued)  
    for program portability [3-8](#)  
    for the CRE [3-5](#)  
    mixed-language programs [3-8](#)  
    rules [3-6](#)  
    run-time libraries [3-7](#)  
Bit manipulation [7-24](#)

## C

C  
    binding [3-5](#)  
    fopen\_std\_file routine [2-28](#)  
    main routine [2-10](#)  
    messages [10-26](#)  
    program initialization [2-12](#), [2-13](#)  
    program termination [2-16](#)  
    requesting heap space [2-40](#), [2-43](#)  
    run-time environment [1-1](#)  
    run-time library [3-7](#)  
    stdfiles directive [2-13](#)  
    traps [2-52](#)  
    using the CRE [3-4](#)  
    \$RECEIVE and [2-36](#)  
c89 utility [4-1](#)  
C8LIB [3-7](#)  
Catastrophic error, CRE handling of [2-49](#)  
Changing  
    environment information [5-8](#)  
    environment values [5-7](#)  
Circumventing the CRE [2-67](#)  
CLIB [3-7](#)  
CLU library  
    see Common Language Utility library  
CLUDECS [1-4](#), [5-2](#), [5-3](#)  
CLURDECS [1-4](#), [5-3](#)

[CLU\\_Process\\_Create\\_function 9-1](#)  
[CLU\\_Process\\_File\\_Name\\_function 5-2, 9-1, 9-12](#)  
**COBOL**  
     binding [3-5](#)  
     locating file connectors [9-1](#)  
     main routine [2-10](#)  
     messages [10-26/10-31](#)  
     program initialization [2-12, 2-13](#)  
     program termination [2-16](#)  
     requesting heap space [2-40, 2-43](#)  
     run-time environment [1-1](#)  
     run-time library [3-7](#)  
     SMU functions [9-1](#)  
     traps [2-53](#)  
     using the CRE [3-4](#)  
     \$RECEIVE and [2-35](#)  
**COMMON Binder group 3-5**  
**Common Language Utility library**  
     see also Saved Message Utility  
     compared to CRE [9-1](#)  
     compared to the CRE [5-1](#)  
     function  
         binding [5-2](#)  
         CLU\_Process\_File\_Name\_ [9-1, 9-12](#)  
         compiling [5-2](#)  
         definition [5-1](#)  
     native CRE and [2-6](#)  
**Compiling**  
     ENV directive [3-4](#)  
     for a language-specific run-time environment [3-4](#)  
     for the CRE [3-4](#)  
     for the native CRE [4-1](#)  
**Connection 2-17, 2-18**  
**Converting applications 3-1**  
**Cos 7-7**  
**Cosh 7-7**

**CRE**  
     advantages of using [1-8](#)  
     basic control block [3-5](#)  
     circumventing [2-67](#)  
     compared to Common Language Utility library [9-1](#)  
     compared to the Common Language Utility library [5-1](#)  
     compiling for [1-7](#)  
     data block  
         CRE\_GLOBALS [3-5](#)  
         CRE\_HEAP [3-5](#)  
         master control block [3-5](#)  
         MCB [3-5](#)  
     initialization  
         see also TAL\_CRE\_INITIALIZER\_procedure  
         errors [2-14](#)  
         in general [2-9](#)  
         tasks [2-10](#)  
     language support [1-8](#)  
     master control block [3-5](#)  
     messages  
         see Messages  
     resources [2-1](#)  
     run-time heap [3-5](#)  
     service functions [6-1](#)  
     services [2-1](#)  
     termination [2-7, 2-8, 2-15](#)  
     trap handler [2-50](#)  
     CRE service messages [10-6/10-11](#)  
**CREDECS 6-1**  
     in general [1-4, 6-1](#)  
     using [3-4, 4-3](#)  
**CRELIB**  
     functions, sourcing-in [3-4, 4-3](#)  
     in general [3-8](#)  
**CRERDECS 1-4, 2-8, 6-1**  
**CRE/RTL 1-3**  
**CRE\_ prefixes 2-57, 2-58**



[CRE\\_File\\_Close\\_ 6-5](#)  
[CRE\\_File\\_Control\\_ 6-6](#)  
[CRE\\_File\\_Input\\_ 6-8](#)  
[CRE\\_File\\_Message\\_ 6-9](#)  
[CRE\\_File\\_Open\\_ 6-11](#)  
[CRE\\_File\\_Output\\_ 6-20](#)  
[CRE\\_File\\_Retrycheck\\_ 6-22](#)  
[CRE\\_File\\_Setmode\\_ 6-23](#)  
[CRE\\_Getenv\\_ 5-9, 6-1](#)  
[CRE\\_GLOBALS 3-5](#)  
[CRE\\_HEAP 3-5](#)  
[CRE\\_Log\\_GetPrefix\\_ 6-45](#)  
[CRE\\_Log\\_Message\\_ 6-25](#)  
[CRE\\_Putenv\\_ 5-9, 6-2](#)  
[CRE\\_Receive\\_Open\\_Close\\_ 6-31](#)  
[CRE\\_Receive\\_Read\\_ 6-38](#)  
[CRE\\_Receive\\_Write\\_ 6-41](#)  
[CRE\\_Spool\\_Start\\_ 6-27](#)  
[CRE\\_Stacktrace\\_ 6-45](#)  
[CRE\\_Terminator\\_ 6-42](#)  
[CRE\\_Terminator\\_ procedure 2-16](#)

## D

[Data blocks 3-5](#)  
[Data types A-1](#)  
[Decimal conversion function](#)  
     [Decimal\\_to\\_Int 7-26](#)  
     [Int\\_to\\_Decimal 7-27](#)  
[Decimal\\_to\\_Int 7-26](#)  
[Declarations file 3-4, 4-3](#)  
[Default signal handler 2-3](#)  
[DEFINE command, OSS 2-45](#)  
[DEFINE command, TACL product 2-45](#)  
[delete\(\) function 2-43](#)  
[Deleting environment information 5-9](#)

## E

[EMBEDDED Binder environment option 3-5](#)  
[EMS events 2-2](#)

[emulate A-3](#)  
[ENV directive 1-7, 3-4, 4-1](#)  
     [Binder groups 3-5](#)  
     [default values 3-3, 4-1](#)  
[ENV pragma](#)  
     [see ENV directive](#)  
[environ array 2-12, 5-9](#)  
[Environment](#)  
     [environ array 5-9](#)  
     [function](#)  
         [CRE\\_Getenv\\_ 5-9, 6-1](#)  
         [CRE\\_Putenv\\_ 5-9, 6-2](#)  
     [information](#)  
         [changing 5-8](#)  
         [deleting 5-9](#)  
         [getting 5-7](#)  
         [values, changing 5-7](#)  
[Erase on free, heap management attribute 2-46](#)  
[Error](#)  
     [catastrophic, CRE handling of 2-49](#)  
     [CRE initialization 2-14](#)  
     [CRE termination 2-16](#)  
     [math standard function 2-48](#)  
     [program logic 2-48](#)  
[Error messages](#)  
     [see Messages](#)  
[Event management service 2-2](#)  
[Exception-handling function](#)  
     [CRE\\_Log\\_GetPrefix\\_ 6-45](#)  
     [CRE\\_Stacktrace\\_ 6-45](#)  
[EXECUTION-LOG, PARAM](#)  
     [program initialization and 2-33](#)  
     [standard input and 2-29, 2-30](#)  
     [standard log and 2-33, 2-34](#)  
[EXECUTION-LOG, PARAM](#)  
     [standard output and 2-31/2-32](#)  
[Exp 7-8](#)  
[Extended stack 2-39](#)

## F

Fault tolerance, CRE

See Process Pairs

File connectors

identifying [5-2](#)

locating [5-2](#)

native CRE and [2-6](#)

processing [9-1](#), [9-12](#)

File sharing

see Sharing standard files

File-sharing function

CRE\_File\_Close\_ [6-5](#)

CRE\_File\_Control\_ [6-6](#)

CRE\_File\_Input\_ [6-8](#)

CRE\_File\_Message\_ [6-9](#)

CRE\_File\_Open\_ [6-11](#)

CRE\_File\_Output\_ [6-20](#)

CRE\_File\_Retrycheck\_ [6-22](#)

CRE\_File\_Setmode\_ [6-23](#)

CRE\_Log\_Message\_ [6-25](#)

CRE\_Receive\_Open\_Close\_ [6-31](#)

CRE\_Receive\_Read\_ [6-38](#)

CRE\_Receive\_Write\_ [6-41](#)

CRE\_Spool\_Start\_ [6-27](#)

File\_Close\_ [6-5](#)

File\_Control\_ [6-6](#)

File\_Input\_ [6-8](#)

File\_Message\_ [6-9](#)

File\_Open\_ [6-11](#)

File\_Output\_ [6-20](#)

File\_Retrycheck\_ [6-22](#)

File\_Setmode\_ [6-23](#)

fopen\_std\_file, C library function [2-28](#)

FORTLIB [3-7](#)

FORTTRAN

binding [3-5](#)

HIGHCONTROL directive [3-5](#)

locating file connectors [9-1](#)

main routine [2-10](#)

FORTTRAN (continued)

messages [10-25/10-31](#)

program initialization [2-12](#), [2-13](#)

program termination [2-16](#)

requesting heap space [2-40](#), [2-43](#)

run-time environment [1-1](#)

run-time library [3-7](#)

SMU functions [9-1](#)

standard log [2-33](#)

traps [2-54](#)

using the CRE [3-4](#)

\$RECEIVE and [2-35](#)

FORTSYS [3-7](#)

free() function [2-43](#)

FULL Binder environment option [3-5](#)

Function parameter message [10-14](#)

Function parameter messages [10-17](#)

Function, standard, using [2-56](#)

## G

Get environment variable [6-1](#)

getenv [5-9](#)

Getting environment information [5-7](#)

Guardian environment

see OSS environment, compared to  
Guardian environment

## H

HEAP [2-38](#)

Heap

sharing [2-39](#), [2-42](#)

space, requesting [2-40](#), [2-43](#)

statistics [2-40](#), [2-41](#)

HEAP data block

Heap management attributes [2-44](#)

Heap management messages [10-12/10-14](#)

Heap managers, native [2-43](#)

HEAP, PARAM not supported [1-8](#), [2-40](#)

**I**

INFILE in startup message [2-29](#)  
 Initialization in CRE [2-10](#)  
 Input/output messages [10-18/10-24](#)  
 Inspect, using with the CRE [2-62](#)  
 Int\_to\_Decimal [7-27](#)  
 I/O  
     with the CRE [2-19](#)  
     without the CRE [2-17](#)

**L**

Language support [1-8](#)  
 Language-specific run-time environment  
     definition [1-1](#)  
 Language-specific run-time libraries  
     definition [1-1](#)  
 LIBRARY Binder environment option [3-5](#)  
 LIBSPACE Binder environment option [3-5](#)  
 Ln [7-8](#)  
 Log10 [7-9](#)  
 Logic errors, CRE handling of [2-48](#)  
 Log\_GetPrefix\_ [6-45](#)  
 Log\_Message\_ [6-25](#)  
 Lower [7-10](#)

**M**

Main routine  
     designating [2-9](#)  
     standard files [2-28](#)  
 malloc() function [2-43](#)  
 Manipulation function for 64-bit data  
     bit manipulation [7-24](#)  
     Remainder [7-25](#)  
 Master control block [3-5](#)  
 Math function  
     Arccos [7-4](#)  
     Arcsin [7-5](#)  
     Arctan [7-5](#)  
     Arctan2 [7-6](#)

## Math function (continued)

    arithmetic overflow handling and [7-1](#)  
     Cos [7-7](#)  
     Cosh [7-7](#)  
     errors [2-48](#)  
     Exp [7-8](#)  
     Ln [7-8](#)  
     Log10 [7-9](#)  
     Lower [7-10](#)  
     Mod [7-11](#)  
     Normalize [7-12](#)  
     Odd [7-13](#)  
     Positive\_Diff [7-13](#)  
     Power [7-15](#)  
     Power2 [7-16](#)  
     Random\_Next [7-17](#)  
     Random\_Set [7-17](#)  
     Round [7-17](#)  
     Sign [7-18](#)  
     Sin [7-19](#)  
     Sinh [7-19](#)  
     Split [7-20](#)  
     Sqrt [7-20](#)  
     Tan [7-21](#)  
     Tanh [7-21](#)  
     Truncate [7-22](#)  
     Upper [7-22](#)  
 Math function messages [10-15/10-16](#)  
 MCB [3-5](#)  
 MCB pointer [2-37](#), [2-42](#), [2-64](#)  
 Memory handling errors with heap  
     manager [2-44](#)  
 Memory organization  
     in general [2-37](#), [2-42](#)  
     in the OSS environment compared to  
     Guardian environment [2-3](#)  
     native CRE [2-41](#)  
     TNS CRE [2-37](#)

## Memory-block function

Memory\_Compare\_ [8-34](#)Memory\_Copy\_ [8-35](#)Memory\_Findchar\_ [8-36](#)Memory\_Move\_ [8-37](#)Memory\_Repeat\_ [8-38](#)Memory\_Set\_ [8-39](#)Memory\_Swap\_ [8-40](#)Memory\_Compare\_ [8-34](#)Memory\_Copy\_ [8-35](#)Memory\_Findchar\_ [8-36](#)Memory\_Move\_ [8-37](#)Memory\_Repeat\_ [8-38](#)Memory\_Set\_ [8-39](#)Memory\_Swap\_ [8-40](#)

## Messages

C [10-26](#)CRE service [10-6/10-11](#)format of [10-2](#)FORTRAN [10-25/10-31](#)function parameter [10-14](#), [10-17](#)heap management [10-12/10-14](#)input/output [10-18/10-24](#)mapping message numbers between  
run-time environments [10-26](#)mapping original to CRE  
numbers [10-26](#)math function [10-15/10-16](#)parts [5-6](#), [5-7](#)trap [10-3/10-5](#)Mixed-environment programming [1-8](#)Mixed-language programming [1-8](#), [2-8](#)Mod [7-11](#)**N**

## Native CRE

arithmetic overflow handling [7-1](#)CRE\_GETENV\_ [6-2](#)CRE\_PUTENV\_ [6-3](#)decimal conversion functions and [7-25](#)

## Native CRE (continued)

file-sharing functions [6-4](#)memory block functions [8-33](#)RTLREDCS file [7-1](#)run-time diagnostic messages [10-25](#)SMU functions [9-17](#)standard math functions [7-3](#)string functions [8-3](#)NATIVE CRE/RTL [1-3](#)NEUTRAL Binder group [3-5](#)new() function [2-43](#)Normalize [7-12](#)**O**Odd [7-13](#)OLD Binder group [3-5](#)OLD binder group [3-5](#)

## OSS environment

compared to Guardian environment

in general [2-2/2-6](#)memory organization [2-3](#)process pairs [2-6](#)program initialization [2-4](#)shared message utility routines [2-6](#)standard files [2-2](#)standard functions [2-5](#)traps and exceptions [2-3](#)\$RECEIVE and [2-3](#)error reporting and [2-54](#)memory organization and [2-37](#), [2-41](#)process pairs and [2-59](#)program initialization and [2-9](#)program termination and [2-15](#)sharing standard files and [2-17](#)signals and [2-3](#), [2-48](#)spooler collectors and [2-36](#)traps and exceptions and [2-48](#)\$RECEIVE and [2-34](#)OUTFILE in startup message [2-29](#)

## P

PARAM command  
     EXECUTION-LOG  
         program initialization and [2-33](#)  
     SAVE-ENVIRONMENT  
         environ array and [5-9](#)  
         using [5-7](#)

PARAM message  
     see also PARAM command  
     changing [9-29](#)  
     creating [9-29](#)  
     deleting [9-27](#)  
     parts [5-7](#)  
     retrieving [9-28](#)

Positive\_Diff [7-13](#)

Power [7-15](#)

Power2 [7-16](#)

Process pairs [2-59](#)  
     C [2-60](#)  
     CRE control of [2-59](#)  
     in general [2-59](#)  
     in the OSS environment compared to  
     Guardian environment [2-6](#)  
     language support for [2-60](#)  
     native CRE and [2-6](#)  
     requirements for using [2-59](#)  
     status codes [2-61](#)

Program initialization  
     in general [2-9](#)  
     in the OSS environment compared to  
     Guardian environment [2-4](#)  
     native CRE [2-4](#), [2-13](#)  
     TNS CRE [2-4](#), [2-10](#)

Program logic errors, CRE handling of [2-48](#)

Program termination  
     compared to Guardian environment  
         program termination [2-4](#)  
     in general [2-15](#)

Program termination (continued)  
     in the OSS environment compared to  
     Guardian environment [2-4](#)

Programs, mixed environment [1-8](#)

pTAL  
     compiling modules for native CRE [4-2](#)  
     linking for the native CRE [4-3](#)  
     traps [2-54](#)  
     using with the CRE [2-8](#)

PTALLIB [2-8](#)

pTAL\_CRE\_INITIALIZER\_ procedure [2-15](#)

Put environment variable [6-2](#)

putenv [5-9](#)

## R

Random\_Next [7-17](#)

Random\_Set [7-17](#)

Receive function  
     CRE\_Receive\_Open\_Close\_ [6-31](#)  
     CRE\_Receive\_Read\_ [6-38](#)  
     CRE\_Receive\_Write\_ [6-41](#)

Receive\_Open\_Close\_ [6-31](#)

Receive\_Read\_ [6-38](#)

Receive\_Write\_ [6-41](#)

Remainder [7-25](#)

Round [7-17](#)

RTLDECH [2-45](#)

RTLDECS [1-4](#), [7-1](#)

RTLDECS, using [3-4](#), [4-3](#)

RTLDECS [1-4](#), [2-45](#)

RTL\_ prefixes [2-57](#), [2-58](#)

Run-time environment  
     determining [3-7](#)  
     selecting [1-7](#)

Run-time libraries [3-7](#)

## S

SAVE compiler directive [5-5](#)

## Saved Message Utility

see also Common Language Utility library

description [5-3](#), [9-17](#)

environment information [5-7](#)

environment values [5-7](#)

functions

SMU\_Assign\_CheckName\_ [9-18](#)

SMU\_Assign\_Delete\_ [9-19](#)

SMU\_Assign\_GetText\_ [9-21](#)

SMU\_Assign\_GetValue\_ [9-22](#)

SMU\_Assign\_PutText\_ [9-23](#)

SMU\_Assign\_PutValue\_ [9-25](#)

SMU\_Message\_CheckNumber\_ [9-26](#)

SMU\_Param\_Delete\_ [9-27](#)

SMU\_Param\_GetText\_ [9-28](#)

SMU\_Param\_PutText\_ [9-29](#)

SMU\_Startup\_Delete\_ [9-30](#)

SMU\_Startup\_GetText\_ [9-31](#)

SMU\_Startup\_PutText\_ [9-33](#)

messages

content [5-6](#)

saving [5-5](#)

services provided by [5-5](#)

using from COBOL [5-5](#)

using from FORTRAN [5-5](#)

using from TAL [5-5](#)

## Saved Message Utility routines

in the OSS environment compared to Guardian environment [2-6](#)

## SAVE-ENVIRONMENT, PARAM

environ array and [5-9](#)

using [5-7](#)

## Saving messages for SMU functions [5-5](#)

## Service functions, calling [6-1](#)

## Sharing standard files [2-17](#)

## Sign [7-18](#)

## Signals [2-3](#), [2-48](#)

## Sin [7-19](#)

## Sinh [7-19](#)

## Sixty-four-bit logical operation

bit manipulation [7-24](#)

Remainder [7-25](#)

## SMU functions

see Saved Message Utility, functions

SMU\_Assign\_CheckName\_ function [9-18](#)

SMU\_Assign\_Delete\_ Function [9-19](#)

SMU\_Assign\_GetText\_ function [9-21](#)

SMU\_Assign\_GetValue\_ function [9-22](#)

SMU\_Assign\_PutText\_ function [9-23](#)

SMU\_Assign\_PutValue\_ function [9-25](#)

SMU\_Message\_CheckNumber\_ function [9-26](#)

SMU\_Param\_Delete\_ function [9-27](#)

SMU\_Param\_GetText\_ function [9-28](#)

SMU\_Param\_PutText\_ function [9-29](#)

SMU\_Startup\_Delete\_ function [9-30](#)

SMU\_Startup\_GetText\_ function [9-31](#)

SMU\_Startup\_PutText\_ function [9-33](#)

Sourcing-in CRELIB functions [3-4](#), [4-3](#)

## Split [7-20](#)

Spooler collector, using [2-36](#)

Spool\_Start\_ [6-27](#)

## Sqrt [7-20](#)

Stacktrace\_ [6-45](#)

## Standard files

C [2-28](#)

CRE routines [2-27](#)

determining when open [2-28](#)

in the OSS environment compared to Guardian environment [2-2](#)

overview [2-17](#)

processes [2-28](#)

sharing access to [2-17](#)

standard input [2-29](#)

standard log [2-33](#)

standard output [2-31](#)

terminals [2-28](#)

with the CRE [2-18](#)

without the CRE [2-17](#)

## Standard functions

in the OSS environment compared to  
Guardian environment [2-5](#)  
using [2-56](#)

Standard input [2-29](#)

Standard log

FORTTRAN and [2-33](#)  
in general [2-33](#)

Standard output [2-31](#)

Startup message

changing [9-33](#)  
creating [9-33](#)  
deleting [9-30](#)  
in general [2-29](#)  
retrieving [9-31](#)

Statistics, heap [2-41](#)

Stcarg [8-4](#)

Stccpy [8-5](#)

Stcd\_I [8-6](#)

Stcd\_L [8-7](#)

Stch\_I [8-8](#)

Stci\_D [8-9](#)

Stcpm [8-10](#)

Stcpma [8-11](#)

Stcu\_D [8-12](#)

STDERR and standard log [2-33](#)

stdfiles C directive [2-13](#)

STDIN, not recognized [2-30](#)

STDOUT, not recognized [2-32](#)

Stpblk [8-13](#)

Stpsym [8-14](#)

Stptok [8-15](#)

Strcat [8-16](#)

Strchr [8-17](#)

Strcmp [8-18](#)

Strcpy [8-19](#)

Strcspn [8-20](#)

String function

Atof [8-3](#)  
Atoi [8-3](#)  
Atol [8-3](#)

## String function (continued)

Stcarg [8-4](#)

Stccpy [8-5](#)

Stcd\_I [8-6](#)

Stcd\_L [8-7](#)

Stch\_I [8-8](#)

Stci\_D [8-9](#)

Stcpm [8-10](#)

Stcpma [8-11](#)

Stcu\_D [8-12](#)

Stpblk [8-13](#)

Stpsym [8-14](#)

Stptok [8-15](#)

Strcat [8-16](#)

Strchr [8-17](#)

Strcmp [8-18](#)

Strcpy [8-19](#)

Strcspn [8-20](#)

Strlen [8-21](#)

Strncat [8-22](#)

Strncmp [8-23](#)

Strncpy [8-24](#)

Strpbrk [8-25](#)

Strrchr [8-26](#)

Strspn [8-27](#)

Strstr [8-27](#)

Strtod [8-28](#)

Strtol [8-29](#)

Strtoul [8-30](#)

Substring\_Search [8-32](#)

table of [8-1](#)

Strlen [8-21](#)

Strncat [8-22](#)

Strncmp [8-23](#)

Strncpy [8-24](#)

Strpbrk [8-25](#)

Strrchr [8-26](#)

Strspn [8-27](#)

Strstr [8-27](#)

Strtod [8-28](#)

Strtol [8-29](#)  
 Strtoul [8-30](#)  
 Substring\_Search [8-32](#)  
 System library [3-7](#)

## T

### TAL

binding [3-5](#)  
 ENV NEUTRAL directive [3-6](#)  
 INITIALIZER system procedure [2-6](#)  
 main routine [2-10](#)  
 program initialization [2-12](#), [2-13](#)  
 program termination [2-16](#)  
 requesting heap space [2-40](#), [2-43](#)  
 run-time environment [1-1](#)  
 run-time library [3-7](#)  
 selecting ENV directive [2-6](#)  
 Sequential I/O (SIO) routines [2-6](#)  
 SMU functions [9-1](#)  
 TALLIB [3-7](#)  
 traps [2-54](#)  
 user data segment [2-6](#)  
 using the CRE [3-4](#)  
 using with the CRE [2-6](#)  
 \$RECEIVE and [2-35](#)

TALLIB [1-4](#), [3-7](#)

TAL\_CRE\_INITIALIZER\_ [5-5](#)

TAL\_CRE\_INITIALIZER\_ procedure [2-7](#),  
[2-15](#), [3-7](#)

Tan [7-21](#)

Tanh [7-21](#)

Termination function,  
 CRE\_Terminator\_ [2-16](#), [6-42](#)

Terminator\_ [6-42](#)

TNS and native programs [1-8](#)

### Traps

see also Signals  
 arithmetic overflow [2-49](#)  
 ARMTRAP [2-51](#)  
 C routines [2-52](#)

### Traps (continued)

COBOL routines [2-53](#)  
 CRE handling of [2-49](#)  
 CRE trap handler [2-50](#)  
 FORTRAN routines [2-54](#)  
 hardware [2-49](#)  
 in the OSS environment compared to  
 Guardian environment [2-3](#)  
 language-specific, handling [2-52](#)  
 messages [10-3/10-5](#)  
 TAL [2-54](#)

Truncate [7-22](#)

Type suffixes [2-59](#)

## U

Upper [7-22](#)

User heap [2-39](#), [2-42](#)

User library routines [3-3](#)

## V

Visual Inspect, using with the CRE [2-62](#)

## W

White space, defined [8-1](#)

## Z

ZCREDLL [1-4](#)

ZCRESRL [1-4](#)

## Special Characters

### \$RECEIVE

C routines [2-36](#)  
 COBOL routines [2-35](#), [2-36](#)  
 COBOL routines and [2-36](#)  
 CRE initialization [2-10](#)  
 FORTRAN routines [2-35](#), [2-36](#)  
 FORTRAN routines and [2-36](#)



**\$RECEIVE** (continued)

in the OSS environment compared to  
Guardian environment [2-3](#)

messages received from [2-35](#)

OSS environment and [2-34](#)

program initialization [2-10](#), [2-35](#)

reading [2-35](#)

TAL routines [2-35](#), [2-36](#)

TAL routines and [2-36](#)

using [2-34](#)

**\_ERASE\_ON\_FREE\_ DEFINE** [2-45](#)

