System Software Library

TACL Programming Guide

Abstract	This manual describes the Tandem Advanced Command Language (TACL) and provides information and examples for creating TACL programs.
Part Number	107365
Edition	Second
Published	December 1994
Product Version	TACL D30
Release ID	D30.00
Supported Releases	This manual supports D30.00 and all subsequent releases until otherwise indicated in a new edition.

Tandem Computers Incorporated

Document History

Edition	Part Number	Product Version	Earliest Supported Release	Published
First	085797	TACL C20	N/A	November 1992
Update	086700	TACL D10	N/A	February 1993
Second	107365	TACL D30	D30.00	December 1994

New editions incorporate any updates issued since the previous edition.

A plus sign (+) after a release ID indicates that this manual describes function added to the base release, either by an interim product modification (IPM) or by a new product version on a .99 site update tape (SUT).

Ordering Information

For manual ordering information: domestic U.S. customers, call 1-800-243-6886; international customers, contact your local sales representative.

Document Disclaimer

Information contained in a manual is subject to change without notice. Please check with your authorized Tandem representative to make sure you have the most recent information.

Export Statement

Export of the information contained in this manual may require authorization from the U.S. Department of Commerce.

Examples

Examples and sample programs are for illustration only and may not be suited for your particular purpose. Tandem does not warrant, guarantee, or make any representations regarding the use or the results of the use of any examples or sample programs in any documentation. You should verify the applicability of any example or sample program before placing the software into productive use.

U.S. Government Customers

FOR U.S. GOVERNMENT CUSTOMERS REGARDING THIS DOCUMENTATION AND THE ASSOCIATED SOFTWARE:

These notices shall be marked on any reproduction of this data, in whole or in part.

NOTICE: Notwithstanding any other lease or license that may pertain to, or accompany the delivery of, this computer software, the rights of the Government regarding its use, reproduction and disclosure are as set forth in Section 52.227-19 of the FARS Computer Software-Restricted Rights clause.

RESTRICTED RIGHTS NOTICE: Use, duplication, or disclosure by the Government is subject to the restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 52.227-7013.

RESTRICTED RIGHTS LEGEND: Use, duplication or disclosure by the Government is subject to restrictions as set forth in paragraph (b)(3)(B) of the rights in Technical Data and Computer Software clause in DAR 7-104.9(a). This computer software is submitted with "restricted rights." Use, duplication or disclosure is subject to the restrictions as set forth in NASA FAR SUP 18-52 227-79 (April 1985) "Commercial Computer Software — Restricted Rights (April 1985)." If the contract contains the Clause at 18-52 227-74 "Rights in Data General" then the "Alternate III" clause applies.

U.S. Government Users Restricted Rights — Use, duplication or disclosure restricted by GSA ADP Schedule Contract.

New and Changed Information

This is the second edition of the *TACL Programming Guide* . This edition documents the following new TACL features:

- A new built-in function, #SETCONFIGURATION, sets the TACL flags that can change the behavior of TACL for a specified TACL image or configure the currently running TACL process.
- A new built-in function, #XLOGON, implements the LOGON command.
- The LOGON command, #CHANGEUSER built-in function, and #XLOGON built-in function support the Safeguard authentication dialog.
- The STATUS command and #XSTATUS built-in function display new process type information.
- CLASS DEFAULT DEFINEs have eight new optional attributes for internationalization support.

(This page left intentionally blank)

Contents

About This Manual xi

Notation Conventions xv

Section 1 An Overview of TACL

Running the Examples in This Manual 1-1

Style Conventions 1-2

- Exceptions to the Style Conventions 1-3
- Conventions Specific to This Manual 1-3

Section 2 Developing TACL Programs

Choosing a Type of Variable 2-1

Defining Program Structure 2-2

- Using Flow Control Functions 2-2
- Nesting TACL Code 2-7
- Saving Levels of Variables 2-8
- Exiting From Programs 2-9

Processing Character Data 2-9

- Line and Character 2-10
- Global Editing Commands 2-13
- Additional Data Manipulation Capabilities 2-16
- Data Types 2-17

Accessing Time Data 2-17

- Timestamp Formats 2-17
- Retrieving a Timestamp 2-18
- Converting a Timestamp 2-19

Accessing Terminals 2-24

- Defining Function Keys 2-24
- Sending Escape Sequences to a Terminal 2-25
- Changing the TACL Prompt 2-29
- Implementing Menus 2-30

Debugging TACL Programs 2-32

- Enabling the TACL Debugger 2-32
- Debugger Commands 2-32
- A Sample Debugging Session 2-33

Section 3 Developing TACL Routines

Processing Arguments 3-1
- How #ARGUMENT Works 3-3
- Using #ARGUMENT 3-4
- Examining the Contents of Arguments 3-9
- Parsing Arguments for a Caller 3-11

Returning Results 3-15

Calling a Routine Recursively 3-16

Exiting From a Routine 3-17

Writing an Exception Handler 3-18
- Types of Exception Handlers 3-19
- Constructing an Exception Handler 3-19
- Creating a Release Exception Handler 3-20
- Creating a Keep Exception Handler 3-24
- Combining Keep and Release Handlers 3-30

Section 4 Accessing Files

#REQUESTER Operation 4-1

Requesting Waited Reads 4-2

Requesting Nowaited Reads 4-4

Requesting Waited Writes 4-6

Requesting Nowaited Writes 4-8

Copying Records Between Files 4-10

Comparing Files 4-12

Listing a File 4-16

Section 5 Initiating and Communicating With Processes

Initiating a Process 5-2
- Using RUN and #NEWPROCESS Options 5-2
- Sending Information at Initiation Time 5-3

Communicating With a Process 5-4
- Using the INLINE Facility 5-6
- Using INV and OUTV 5-14
- Using $RECEIVE 5-21
- Using #SERVER 5-29
- Using Define Process 5-31

Processing Completion Information 5-32
Processing NetBatch Jobs and Completion Codes 5-32
Monitoring Job Status
ENQUIRY 5-35

Section 6 Running TACL as a Server

Running a TACL Process as a Server 6-1
Starting TACL as a Server Process 6-1
Sending Requests to a TACL Server 6-2
Directing Output From TACL 6-4
Running TACL Code as a Server 6-5
Constructing a TACL Server 6-5
Using TACL as a Pathway Server 6-6

Section 7 Using Programmatic Interfaces

Overview of SPI and EMS 7-1
Using SPI 7-4
Defining an SPI Buffer 7-5
Using SPI Functions 7-9
Using EMS 7-12
Communicating With EMS 7-12
Generating an EMS Event 7-13

Section 8 Example of a System Management Program

Monitoring System Operation 8-1

Section 9 Syntax Summary

TACL Commands and Functions 9-1
Built-In Functions and Variables 9-6
STRUCT Declarations 9-14
#SET Summary 9-15
#DELTA Command Summary 9-16

Appendix A Supplemental Information for D-Series Systems

Glossary Glossary–1

Index Index–1

Figures

Figure 2-1. Performing Tasks Within a Loop 2-2
Figure 2-2. Performing a Bubble Sort With Nested #LOOP Statements 2-3
Figure 2-3. Deleting Files in a Subvolume 2-5
Figure 2-4. Processing Macro Arguments 2-8
Figure 2-5. Extracting a Volume Name from a Variable 2-12
Figure 2-6. Retrieving Disk Names From DSAP 2-15
Figure 2-7. Relationships Between System Timestamps and TACL Functions 2-20
Figure 2-8. Relationships Between #FILEINFO Timestamps and TACL Functions 2-21
Figure 2-9. Computing the Current Day 2-22
Figure 2-10. Converting Timestamps 2-23
Figure 2-11. Sending Special Characters to a Screen 2-25
Figure 2-12. Displaying a Screen of Text 2-27
Figure 2-13. Locking a Terminal 2-28
Figure 2-14. Displaying a Menu 2-30
Figure 2-15. Starting TEDIT From TACL 2-34
Figure 3-1. Processing Arguments 3-5
Figure 3-2. Returning Characters From a Routine 3-9
Figure 3-3. Returning a Set of Characters From a Variable 3-10
Figure 3-4. Searching for Text 3-10
Figure 3-5. Counting Characters in a Variable 3-11
Figure 3-6. Moving Text Between Variables 3-11
Figure 3-7. Assigning Values to Arguments 3-12
Figure 3-8. Sending Arguments to a Parsing Program 3-14
Figure 3-9. Converting Timestamps 3-15
Figure 3-10. Processing Arguments 3-16
Figure 3-11. Processing File Name Arguments 3-17
Figure 3-12. Sample Release Handler Template 3-20
Figure 3-13. Sample Release Handler 3-21
Figure 3-14. Returning Information From a Release Handler 3-22
Figure 3-15. Sample Keep Exception Handler 3-25

Figure 3-16. Sample Command Shell 3-27
Figure 3-17. Using Nested Keep and Release Handlers (Page 1 of 2) 3-31
Figure 4-1 Performing a Waited Read 4-3
Figure 4-2 Performing a Nowaited Read 4-5
Figure 4-3 Reading From a Terminal and Performing a Waited Write 4-7
Figure 4-4 Reading From a Terminal and Performing a Nowaited Write 4-9
Figure 4-5 Copying Records From One File to Another File 4-10
Figure 4-6 Comparing Two Files 4-13
Figure 4-7 Listing a File 4-16
Figure 4-8 TACLLIST Output 4-20
Figure 5-1 Communicating With FUP 5-8
Figure 5-2 Building a Script 5-8
Figure 5-3 Retrieving Output from FUP 5-10
Figure 5-4 Omitting Terminal Output 5-11
Figure 5-5 Deleting PERUSE Jobs 5-12
Figure 5-6 Retrieving the TACL IN File Name 5-15
Figure 5-7 Communicating With FUP Using INV and OUTV 5-17
Figure 5-8 Directing FUP Output to a Log File 5-18
Figure 5-9 Displaying PERUSE Jobs 5-19
Figure 5-10 Sending Messages to a Terminal 5-22
Figure 5-11 Creating CMON Messages 5-25
Figure 5-12 Communicating With FUP Using #SERVER 5-30
Figure 5-13 Checking Completion Codes 5-33
Figure 5-14 Retrieving TACL Output 5-36
Figure 6-1 Starting and Sending Requests to a TACL Server 6-2
Figure 6-2 Running a TACL Program as a Server 6-7
Figure 6-3 Screen COBOL Code That Accesses a TACL Server 6-9
Figure 6-4 Configuring the Pathway Environment 6-11
Figure 7-1 Comparing Two Subsystem IDs 7-9
Figure 7-2 Displaying the EMS Log 7-10
Figure 7-3 Generating an EMS Event 7-13
Figure 8-1 Monitoring System Status 8-2

Tables

Table 2-1. Built-In Functions That Edit Variables by Line 2-10

Table 2-2. Global Editing Commands 2-13

Table 2-3. Data Manipulation Functions 2-16

Table 2-4. Timestamp Conversion Functions 2-19

Table 2-5. _DEBUGGER Command Syntax 2-32

Table 3-1. Functions That Support Arguments 3-3

Table 3-2. Functions That Support Exception Handlers 3-19

Table 3-3. Differences Between Keep and Release Exception Handlers 3-20

Table 4-1 Functions Used With #REQUESTER 4-2

Table 5-1 RUN and #NEWPROCESS Communication Options 5-2

Table 5-2 INLINE Commands and Variables 5-6

Table 5-3 Variables and Commands for INLINE Display 5-9

Table 5-4 Functions and Options Used With INV and OUTV 5-15

Table 5-5 Functions to Use With $RECEIVE 5-21

Table 6-1 Functions That Support Interprocess Communication 6-2

Table 7-1 TACL Functions That Support SPI 7-4

Table 7-2 SPI Token Data Types 7-6

Table 7-3 Functions That Support EMS 7-12

About This Manual

This manual describes the Tandem Advanced Command Language (TACL) and provides information and examples for creating TACL programs.

Audience

This manual is intended for users of TACL who are familiar with TACL commands and built-in functions and who want to create TACL programs.

Organization

This manual contains the following sections:

- ☐ Section 1, "An Overview of TACL," contains an overview of TACL features and a description of the programming conventions used in examples.
- ☐ Section 2, "Developing TACL Programs," describes topics that are common to many TACL programs, whether they are structured as TEXT, MACRO, or ROUTINE variables. Topics include data editing, flow of control, using time data, accessing terminals, handling errors, and debugging TACL programs.
- ☐ Section 3, "Developing TACL Routines," describes how to use TACL constructs that are available only for routines, including the use of #ARGUMENT, #RETURN, and #ROUTINENAME.
- ☐ Section 4, "Accessing Files," provides information and examples that show how to access files from TACL programs.
- ☐ Section 5, "Initiating and Communicating With Processes," provides information and examples that show how to start and communicate with processes from TACL programs.
- ☐ Section 6, "Running TACL as a Server," describes how to create a TACL program that acts as a server to other processes.
- ☐ Section 7, "Using Programmatic Interfaces," provides information and examples for sending SPI and EMS messages.
- ☐ Section 8, "Example of a System Management Program," contains a sample program that monitors system status.
- ☐ Section 9, "Syntax Summary," provides a syntax summary of all TACL functions.
- ☐ Appendix A, "Supplemental Information for D-Series Systems," provides information on D-series features.

Related Reading

The following paragraphs list manuals that are related to the development of TACL programs.

Prerequisites

Introductory material describing the steps involved in using TACL as a command interpreter, as well as using it for defining function keys, writing simple macros, and other basic purposes, is presented in the *Guardian User's Guide* (contains no descriptions of TACL built-in functions and variables). You should read and understand the first four sections of that manual before using this programming guide.

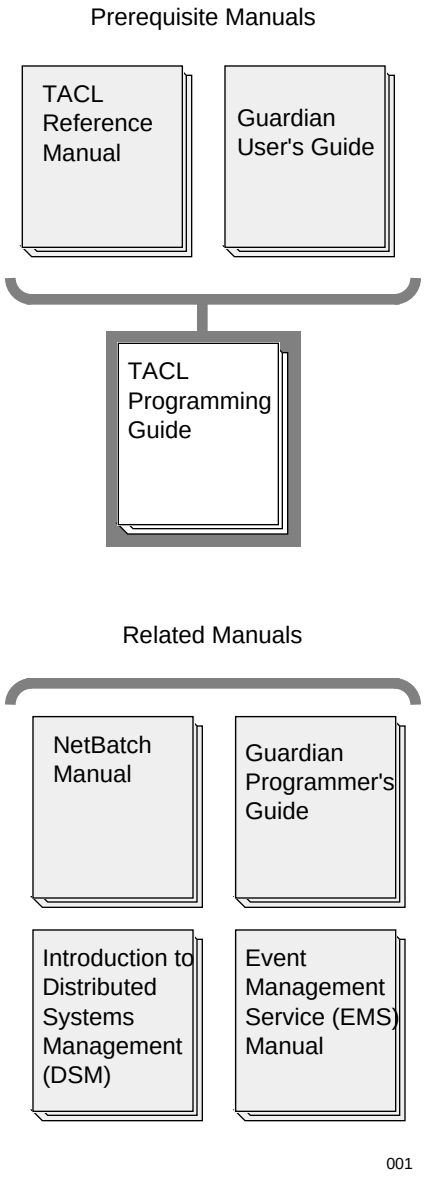
To use this manual, you should be familiar with the syntax and structure of procedural variables (TEXT, MACRO, and ROUTINE), including basic programming concepts and terminology such as “pushing” and “popping” (creating and deleting) variables, the use of arguments, and so on. Elements of the TACL language are described in the *TACL Reference Manual*.

Corequisites Additional sources of information you might want to have available for reference are:

Debug Manual
Event Management Service (EMS) Manual
Enscribe Programmer’s Guide
Expand Network Management Guide
File Utility Program (FUP) Reference Manual
Guardian Programmer’s Guide
Introduction to Distributed Systems Management (DSM)
Distributed Systems Management (DSM) Programming Manual
Inspect Manual
NetBatch Manual
Security Management Guide
System Procedure Errors and Messages Manual
NonStop II and TXP System Operator’s Guide
System Procedure Calls Reference Manual, Volume 1 and 2
Introduction to NonStop Transaction Manager/MP (TM/MP)
ViewPoint Manual

Figure 1 lists the recommended sequence for reading TACL related manuals.

Figure 1. Documentation Road Map



(This page left intentionally blank)

Notation Conventions

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:

```
STATUS
```

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

```
file-name
```

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

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

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:

```
LIGHTS [ ON              ]
       [ OFF             ]
       [ SMOOTH [ num ] ]
```

Note also that TACL uses brackets in commands and functions.

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:

```
#BUILTINS [ / { FUNCTIONS | VARIABLES } / ]
```

Note also that TACL uses braces in comments.

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

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

Note also that TACL uses vertical lines to surround labels in enclosures.

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:

```
#PUSH variable [ [,] variable ] ...

[ + | - ] {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.

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

```
#CHARACTERRULES
```

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

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:

```
PURGE file-name
```

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:

```
$subvol.file-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:

```
#POP variable
   [ [,] variable ]...
```

1 An Overview of TACL

The Tandem Advanced Command Language (TACL) is the standard command interface to the Tandem NonStop kernel. In addition to providing full command interpreter facilities, TACL is a high-level programming language.

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

- ☐ Automate system startup and shutdown procedures.
- ☐ Automate subsystem startup and shutdown procedures; for example, you can use TACL statements to initialize Pathway, the TMF subsystem, Transfer, and other subsystems.
- ☐ Run utilities and issue commands–either with a fixed set of commands or a flexible set that you can tailor at run time.
- ☐ Create a customized environment that simplifies commonly performed tasks for users.
- ☐ Control subsystem operation using the Subsystem Programmatic Interface (SPI).
- ☐ Communicate with the Event Management Service (EMS) and generate EMS messages.

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

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

TACL is extensible; all of the commands supplied by Tandem are implemented as TACL programs. You can add functions as necessary.

In addition, NetBatch requires the use of TACL.

The following paragraphs describe programming style and the use of examples in this manual.

Running the Examples in This Manual

Before running the examples in this manual, set the built-in variable #INFORMAT to TACL, which enables recognition and processing of the TACL special characters ([and], for example); without this step, TACL does not recognize metacharacters as special characters. The function call is:

```
11> #SET #INFORMAT TACL
```

In addition, set the built-in variable #PMSEARCHLIST to include (at least) $SYSTEM.SYSTEM and the keyword #DEFAULTS, which enable the use of implied RUN commands. An example is:

```
12> #SET #PMSEARCHLIST [#DEFAULTS] $SYSTEM.SYSTEM
```

You can add these statements to your TACLCSTM file if you use them frequently.

In cases where TACL provides a built-in function that is similar to a command (such as PUSH and #PUSH), the examples in this manual use the built-in function. Built-in functions are the most basic unit of TACL, and they return results and provide easier programmatic access to error information.

Error checking is as important in TACL programs as it is in programs written in other languages. Possible sources for errors include terminal input and file system operations.

Examples and sample programs are for illustration only and might not be suited for your particular purpose. Tandem does not warrant, guarantee, or make any representations regarding the use or the results of the use of any examples or sample programs in any documentation. You should verify the applicability of any example or sample program before placing the software into productive use.

Note For additional examples, you can view the code for TACL commands by displaying the contents (#OUTVAR *command*) .

Style Conventions

The examples in this manual adhere to the following conventions for clarity and maintainability of programs:

- Built-in functions, commands, variables, and other keywords appear in uppercase:
  - #SET
  - TIME
- User-defined functions, commands, and variables appear in lowercase:
  - `var1`
  - `get_info`
- STRUCT definitions are indented two columns for each level of nesting.
- Similar levels of nested text in #IF, #CASE, and #LOOP statements are indented to the same column. Indentations are in two-space increments. Matching square brackets are in the same column; labels within square brackets start at the same column.
  - Square brackets and labels for #IF and #LOOP statements start at the same column. Conditional text is indented two more spaces. For example:

```
[#IF [timenow] > 12 |THEN|
  #OUTPUT Good afternoon
|ELSE|
  #OUTPUT Good morning
]
```

  - Square brackets for #CASE statements start at the same column. Because #CASE statements can include include several user defined labels; labels are

indented two spaces within the square brackets. Conditional text is indented two spaces past the labels. For example:

```
[#CASE [errornumber]
  |0|
    #OUTPUT [filename] was purged
  |OTHERWISE|
    #OUTPUT [filename] could not be purged
    #OUTPUT Error [errornumber]
]
```

Exceptions to the Style Conventions

In a few situations, the preceding style conventions do not produce optimal results. For example, an #OUTPUT call that continues on a second line includes leading spaces in the display. Therefore, text that continues an #OUTPUT call should be left-justified. For example:

```
[#IF [x] > 0 |THEN|
  #OUTPUT This is a test; the text for this #OUTPUT call &
is longer than a single line.
]
```

The second line of the #OUTPUT call is not indented two spaces.

In such cases, the examples in this manual note the exception and do not follow the style conventions.

Conventions Specific to This Manual

The following additional conventions are used for consistency:

- The COMPUTE command and the #COMPUTE, #IF, and #LOOP built-in functions accept expressions as arguments. When you supply a variable name as all or part of an expression, you can enclose the variable name in square brackets or omit the square brackets. Either way, TACL retrieves the contents of the variable.

  The examples in this manual include the square brackets, to show that the statement uses the contents of the variable. This approach, however, requires slightly more processing by TACL.

- The examples in this manual are restricted to a line length of 62 characters (as opposed to 80 characters for an edit file). There are several function calls in this manual that are longer than 62 characters; these calls are enclosed in square brackets or are joined by an ampersand character:

  ```
  [#SET temp [#CONTIME [#FILEINFO/MODIFICATION/
  [thisfile]]]
  ```

  ```
  #SET temp [#CONTIME [#FILEINFO/MODIFICATION/          &
    [thisfile]]]
  ```

  Lines that have 80 characters or less can fit on one edit file line; if you join these lines in your program, you can omit the surrounding square brackets or the ampersand character.

For more information about expressions, see the *TACL Reference Manual*.

(This page left intentionally blank)

2 Developing TACL Programs

This section describes topics that are common to all types of procedural variables. Topics include:

- ☐ Defining program structure
- ☐ Processing character data
- ☐ Accessing time information
- ☐ Accessing terminals
- ☐ Debugging TACL programs

The *TACL Reference Manual* contains information about TACL statements, programs, and the TACL environment. This information is prerequisite to the topics in this and later sections.

Choosing a Type of Variable

The choice of a type of procedural variable depends on the type of work you plan to do. The variable types can be summarized as follows:

- ☐ A macro is typically used for programs that have limited need for validation of arguments and no need for conditional exits. Within a macro, you can:
  - ☐ Define and access data structures such as text and STRUCT variables.
  - ☐ Compare, move, and manipulate the contents of variables.
  - ☐ Process arguments.
  - ☐ Use TACL built-in functions and commands, including built-in functions that provide conditional execution of code.
  - ☐ Use TACL built-in variables to specify or obtain information about the TACL environment.
- ☐ A routine is the most general and fully functioned type of procedural variable, and is required for programs that handle exceptions (unusual events) or perform complex flow of control operations. If you plan to perform complex argument processing, a routine is recommended.

  Routines provide all of the capabilities that are available from macros, plus they support built-in functions such as #ARGUMENT and #RETURN that are not available to macros. While routines can provide more functionality than macros, they also require more knowledge.

This section contains examples that illustrate the use of macros; except where noted, these techniques also apply to routines.

Section 3, “Developing TACL Routines,” discusses additional features that apply only to TACL routines.

Defining Program Structure

The following paragraphs describe topics that are related to the structure of TACL programs:

- ☐ Using flow control functions: #LOOP, #IF, and #CASE
- ☐ Nesting programs within other TACL programs
- ☐ Saving and restoring levels of variables
- ☐ Exiting from TACL programs

Using Flow Control Functions

The following examples show how #LOOP, #CASE, and #IF statements work.

The macro in Figure 2-1, `copier`, demonstrates two ways to perform an activity in a loop. `Copier` makes six copies of a file named TYPE. (The file TYPE must exist before you run `copier`.) For the first three duplications, `copier` starts a new FUP process during each pass through a loop. For the second three duplications, `copier` loops to prepare a sequence of commands and then passes the commands to FUP. The second method requires one additional variable and one more function call but starts only one FUP process, and is therefore more efficient.

When you run `copier`, your TACL process must be named (using the NAME option with the TACL command). To run `copier`, load the file that contains the macro and then type:

```
copier
```

Figure 2-1. Performing Tasks Within a Loop

```
?SECTION copier MACRO
#FRAME
#PUSH listvar              == List of files to be duplicated
#PUSH sn                   == Serial number

== Less efficient method; multiple processes started
#SET sn 0
[#LOOP |DO|
  FUP DUP type, tmpa[sn]     == Start FUP for each
  #SET sn [#COMPUTE sn + 1]  == command in the loop
|UNTIL| (sn = 3)
]
== More efficient method; one process started
#SET sn 0
[#LOOP |DO|
  #APPEND listvar DUP type, tmpb[sn]
  #SET sn [#COMPUTE sn + 1]
|UNTIL| (sn = 3)
]
FUP/INV listvar/           == Execute FUP only once
#UNFRAME
```

You can define a macro that increments a loop variable (passed as the argument); for example:

```
[#DEF next MACRO |BODY|
  #SET %1% [#COMPUTE %1% + 1]
]
```

Use the macro in Figure 2-2, bubble, with its nested #LOOP statements, to perform a bubble sort. A bubble sort compares numbers and switches their places until the numbers are stored in ascending order.

To run bubble, load its file and supply the number of sort elements as an argument:

```
bubble num
```

Figure 2-2. Performing a Bubble Sort With Nested #LOOP Statements (Page 1 of 2)

```
?SECTION bubble MACRO
#FRAME

#PUSH    i j max ocount element prompt temp
#SETMANY i j max ocount element, 0 0 0 0 0
#SET prompt  Enter Next Number to be Sorted...

== Request the number of elements specified in argument 1 and
== store them in levels of STACK
[#LOOP |WHILE| (element < %1%) |DO|
  #PUSH stack
  #INPUTV stack prompt
  #SET element [#COMPUTE element + 1]
]  {end of INPUT loop}

== Loop once for each element
#SET ocount 1
[#LOOP  |WHILE| (ocount < %1%) |DO|
  #SET i 1
  #SET max [#COMPUTE %1% - ocount + 1]

== Compare each element to its adjacent number; switch places
== if we encounter a smaller number.
  [#LOOP  |WHILE| (i < max) |DO|
    #SET j [#COMPUTE i + 1]
    [#IF  ([stack.i] > [stack.j]) |THEN|
      #SET temp [stack.i]
      #SET stack.i stack.j
      #SET stack.j temp
    ]
  #SET i [#COMPUTE i + 1]
  ]  {end of inner loop}
  #SET ocount [#COMPUTE ocount + 1]
]  == end of outer loop
```

Figure 2-2. Performing a Bubble Sort With Nested #LOOP Statements (Page 2 of 2)

```
== Loop through all variable levels and display contents
#SET element 1
[#LOOP |DO|
  #OUTPUT [stack.[element]]
  #SET element [#COMPUTE element + 1]
|UNTIL| (element > %1%)
]
#UNFRAME
```

`Bubble` requests the specified number of elements and displays the results:

```
29> bubble 3
Enter Next Number to be Sorted...1
Enter Next Number to be Sorted...43
Enter Next Number to be Sorted...5
1
5
43
30>
```

The `bubble` macro does not check the data type of the argument. Therefore, `bubble abc` causes an error:

```
29> bubble abc

[#IF ( ( 0 < abc) )

Expecting a constant
Or NOT
Or a string
(variable does not exist)
Or a number
Or (

30>
```

Use the routine in Figure 2-3, `checkfiles`, with its #IF statements and nested #CASE statements, to perform file maintenance on a subvolume. The routine asks for an alphanumeric starting point in the subvolume. It starts at the next file name, loops through your current subvolume, and displays information about each file in the subvolume.

Note When you run `checkfiles`, you must access your local system and your node name must not be included in your current #DEFAULTS. To remove a node name, if present, enter SYSTEM at the TACL prompt before running `checkfiles`.

When you run checkfiles, the routine displays the following:

```
Where do you want to start (default = beginning of subvol)?
```

To start checking files in the middle of the subvolume, enter a text constant with the desired starting characters. Checkfiles then performs the following steps for each file past the specified starting point in the subvolume:

1. Checkfiles displays the file name and date of last alteration.
2. For an Edit file, checkfiles displays the first ten lines of the file; otherwise, it displays "Not an Edit file; nothing to show you."
3. Checkfiles asks if you want to purge the file (Y or N). For files that are not Edit files, checkfiles also asks if you want to empty the file (E).

Checkfiles stops at the end of the subvolume; to stop earlier, press BREAK.

Figure 2-3. Deleting Files in a Subvolume (Page 1 of 2)

```
?SECTION checkfiles ROUTINE
#FRAME
#PUSH filenm reply prompt volsubvol thisone temp resp
#SET thisone [#DEFAULTS/CURRENT/]
#SET volsubvol [thisone]

#SET prompt   &
  Enter the start file (default = beginning of subvol)?

== Read text from the terminal
#INPUTV filenm prompt
#SET filenm [#NEXTFILENAME [#SHIFTSTRING/UP/[filenm]] ]
[#SET volsubvol
  [#FILEINFO/VOLUME/[filenm]].[#FILEINFO/SUBVOL/[filenm]]
]

== Loop within the same subvolume
[#LOOP |WHILE| ([_COMPAREV volsubvol thisone]) |DO|
  == Display last-altered information
  #SET temp  [#CONTIME [#FILEINFO/MODIFICATION/ [filenm]]]
  #OUTPUT
  #OUTPUT [filenm] last altered [_CONTIME_TO_TEXT [temp]]
  == Get the file code
  [#CASE [#FILEINFO/CODE/ [filenm] ]
  == If an edit file, list the first ten lines, then
  == determine whether the user wants to purge the file
    |101|
      FUP COPY [filenm] , , COUNT 10
      #SET prompt Do you want to purge [filenm] (y/n)?
      #INPUTV reply prompt
```

Figure 2-3. Deleting Files in a Subvolume (Page 2 of 2)

```
          [#IF [#MATCH Y* [#SHIFTSTRING/UP/[reply]]]
          |THEN|
            #SET resp [#PURGE [filenm]]
            [#CASE [resp]
              |0|
                #OUTPUT [filenm] purged
              |OTHERWISE|
                #OUTPUT [filenm] could not be purged
                #OUTPUT Error [resp]
            ] {end #CASE}
          ] {end #IF}

        |OTHERWISE|
          == Not an edit file; determine whether to purge it
          #OUTPUT Not an Edit file; nothing to show you.
          #SET prompt Do you want to purge/empty [filenm] &
            (y/n/e)?
          #INPUTV reply prompt

          [#IF [#MATCH Y* [#SHIFTSTRING/UP/[reply]]] |THEN|
            #SET resp [#PURGE [filenm]]
            [#CASE [resp]
              |0|
                #OUTPUT [filenm] purged
              |OTHERWISE|
                #OUTPUT [filenm] could not be purged
                #OUTPUT Error [resp]
            ] == end #CASE
          ] == end #IF

          [#IF [#MATCH E* [reply]] |THEN|  == empty the file
            FUP PURGEDATA  [filenm]
          ]
      ] == end #CASE

      == Get the next file
      #SET filenm [#NEXTFILENAME [filenm] ]
      [#SET volsubvol
        [#FILEINFO/VOLUME/[filenm]].[#FILEINFO/SUBVOL/[filenm]]
      ] == end #SET
    ] == end #LOOP
    #UNFRAME
```

Note The previous example starts FUP several times; a more efficient way is to start FUP once and send it a series of commands. For information about starting a process and sending it commands, see Section 5, “Initiating and Communicating with Processes.”

Nesting TACL Code

To run another TACL program from within a TACL program, invoke the file name or variable name, as appropriate.

Certain built-in functions can be used only within one type of program (macro or routine). To determine the use of such functions in a nested program, see individual function descriptions in the *TACL Reference Manual*. For example, #ARGUMENT must be used within a routine, but can be used in a macro if the macro is nested within a routine.

To call a program recursively, use %0% (for a macro) or #ROUTINENAME (for a routine) to specify the name of the program. You cannot call a text variable recursively.

The following macro calls itself to display each of its arguments on a separate line:

```
?TACL MACRO
#OUTPUT %1%              == Display current argument
[#IF NOT [#EMPTY %2%] |THEN|
  == Test for additional arguments
  %0% %2 TO *%           == Call self without current
]                        == argument.
```

To run this macro, type the file name from the TACL prompt and supply one or more arguments. The macro displays the arguments you supply. In the following example, the file name that contains the macro is called ARGS:

```
12> args a b c d
a
b
c
d
13>
```

Section 3, “Developing TACL Routines,” contains a macro that calls a nested routine.

Use the macro in Figure 2-4, `defaultvars`, to assign data to a set of empty variables. The macro accepts a space-separated list of variables and nonempty values and calls itself repeatedly until all arguments are processed. To run this macro, load the file that contains the macro definition and then type:

```
12> defaultvars [variable constant [variable constant...] ]
```

Figure 2-4. Processing Macro Arguments

```
?SECTION defaultvars MACRO

== Any more pairs?
[#IF NOT [#EMPTY %1%] |THEN|

  == Is this variable empty?
  [#IF [#EMPTYV %1%] |THEN|

    == The variable is empty; install default value
    #SET %1% %2%
  ]

== Call self again, omitting the current pair.
  %0% %3 TO *%
]
```

The following session shows how `defaultvars` works:

```
15> #PUSH a b c
16>  defaultvars a 3 b 4 c 5

17> #OUTPUTV a
3
18> #OUTPUTV b
4
19> #OUTPUTV c
5
```

Saving Levels of Variables

The #PUSH built-in function creates a new level for a user-defined or built-in variable. If you push a variable twice, TACL creates two levels of the variable. A new level remains in existence until you request a #POP, #UNFRAME, or #RESET FRAMES operation.

The following code redefines the TACL OUT file, retrieves information from the history buffer, saves the history information in a file named HISTFILE in the current subvolume, and then restores the OUT file to its previous setting:

```
?SECTION historysave MACRO
#PUSH #OUT             == Create a new level for #OUT
#SET #OUT histfile     == Set OUT to HISTFILE
#HISTORY               == Retrieve history information
#POP #OUT              == Restore #OUT to its previous value
```

The #FRAME built-in function creates a local environment for variables. The #UNFRAME command restores variables to the state they were in at the time of the last #FRAME operation. For more information about frames, see the *TACL Reference Manual.*

Exiting From Programs

TACL exits from a macro or text variable as soon as it encounters either of the following conditions:

- Successful completion of the code; TACL executes each line and exits when finished.
- Detection of an error condition, in which case TACL restores all variables to the state they were in when the variable was invoked and then exits the variable.

If you use routine variables, you can use the #RETURN built-in function to return conditionally from one or more locations in your code. For more information, see "Returning Results" in Section 3, "Developing TACL Routines."

Processing Character Data

When writing a TACL program, you might need to examine or modify the contents of variables. Such tasks include:

- Constructing text strings for input to processes, files, or devices
- Analyzing process output
- Analyzing results of functions

For example, whenever you use RUN or #NEWPROCESS to initiate a process from TACL, you can direct output from the process to a variable:

- The OUT option directs program output to a file.
- The OUTV option stores program output into a variable for later use.

Section 5, "Initiating and Communicating With Processes," describes process initiation.

TACL supports several commands and functions that manipulate characters and lines of characters within variables:

- Commands, typically used for interactive work, perform editing operations on one or more lines in a variable.
- Built-in functions perform a single operation, referencing text by character position or line number.

#DELTA, the low-level character editor, provides text editing capabilities similar to those provided by the character and line oriented built-in functions. #DELTA is complex; the newer #CHARxxx and #LINExxx built-in functions are easier to use.

STRUCT variables, or structures, allow you to define a set of elements and access the elements by name. STRUCT variables support a range of data types. Structures are helpful when communicating with processes such as $CMON, and are required when communicating with the Subsystem Programmatic Interface (SPI) and the Event Management Service (EMS).

For additional information about #DELTA and STRUCT variables, see the *TACL Reference Manual*. For examples showing the use of STRUCT variables with SPI and EMS, see Section 7, "Using Programmatic Interfaces."

The following paragraphs describe how to use string manipulation functions and commands.

Note Variable levels that contain TACL code contain special internal multicharacter representations of [, |, and]. When you use character oriented functions, be aware that these representations are counted as multiple characters; they contain unprintable characters that are subject to change from one release of TACL to another.

Line and Character Built-In Functions

The functions in Table 2-1 operate on the contents of variables; each of these functions performs an action and/or returns a result. There are two types of functions—one accepts a character address; the other accepts a line address. Table 2-1 lists both types of built-in functions; a dash indicates that there is no equivalent function or command.

Table 2-1. Built-In Functions That Edit Variables by Line

Character Address Function	Line Address Function	Description
#CHARADDR	#LINEADDR	Converts between a character address and a line address.
#CHARBREAK	#LINEBREAK	Inserts a line break at the specified character or line address.
#CHARCOUNT	#LINECOUNT	Counts characters or lines.
#CHARDEL	#LINEDEL	Deletes consecutive characters or lines.
#CHARFIND	#LINEFIND	Finds the address of specified text, searching forward.
#CHARFINDV	#LINEFINDV	Finds the address of a specified string, searching forward.
#CHARFINDR	#LINEFINDR	Finds the address of specified text, searching backward.
#CHARFINDRV	#LINEFINDRV	Finds the address a specified string, searching backward.
#CHARGET	#LINEGET	Returns a copy of consecutive characters or lines.
#CHARGETV	#LINEGETV	Places a copy of consecutive characters or lines in the variable.
#CHARINS	#LINEINS	Inserts lines of text at a specified address in the variable.
#CHARINSV	#LINEINSV	Inserts a string at a specified address in the variable.
—	#LINEJOIN	Joins two lines.

A character address specifies a particular character within a variable, counting from the first character, whose character address is 1. Each end-of-line character (except the last one in the variable) counts as one character. If a specified character address is greater than the number of characters in the variable, that address is considered to be equivalent to the address of the character that appears after the last character in the variable.

A line address specifies a particular line within a variable, counting from the first line, whose line address is 1. If a specified line address is greater than the number of lines in the variable, that address is considered to be equivalent to the address of the line that appears after the last line in the variable. If the variable changes, the assumed line numbers will be different on the next operation. Unlike line numbers in an edit-format file, these assumed line numbers are not saved.

For example, a variable called `sample` contains three lines of text:

```
10> #PUSH sample
11> #APPEND sample This variable is called "sample."
12> #APPEND sample It contains three lines of text.
13> #APPEND sample The contents will be edited by built-in
functions.

14> #OUTPUTV sample

This variable is called "sample."
It contains three lines of text.
The contents will be edited by built-in functions.

15>
```

To retrieve the number of lines, enter:

```
11> #LINECOUNT sample

#LINECOUNT sample expanded to:
3
12>
```

To retrieve the number of characters, enter:

```
12> #CHARCOUNT sample

#CHARCOUNT sample expanded to:
117
13>
```

To find the line address of the first line that contains “text,” starting at line 1, enter:

```
13> #LINEFINDV sample 1 "text"

#LINEFINDV sample 1 "text" expanded to:
2
14>
```

To find the character position of the first occurrence of the string “text,” starting at character position 1, enter:

```
14> #CHARFINDV sample 1 "text"

#CHARFINDV sample 1 "text" expanded to:
62
15>
```

Use the macro in Figure 2-5, `volname`, to extract a volume name from a variable named `mylist`. To use this macro, load the file that contains the macro definition and enter:

```
volname
```

When you invoke this macro, it displays the following:

```
15> volname
The volume name is $DATA
16>
```

Figure 2-5. Extracting a Volume Name from a Variable

```
?SECTION volname MACRO
#FRAME
#PUSH mylist begin end vol

== Specify the contents of mylist:
#SET mylist The disk file name is \SYS1.$DATA.SVOL.NAME1.

== Search for a dollar sign:
#SET begin [#CHARFINDV mylist 1 "$"]

== Search for the first period following the dollar sign:
#SET end [#CHARFINDV mylist [begin] "."]

== Retrieve the characters between "$" and ".":
#SET vol [#CHARGET mylist begin TO [#COMPUTE end - 1]]
#OUTPUT The volume name is vol
#UNFRAME
```

For examples showing the use of these functions for argument processing in routines, see “Processing Arguments” in Section 3, “Developing TACL Routines.”

Global Editing Commands

Use the commands in Table 2-2 to perform editing operations on the entire contents of a variable or a range of lines within the variable.

Table 2-2. Global Editing Commands

Command	Description
VCHANGE	Changes all occurrences of one string to another string in a range of consecutive lines in a variable. VCHANGE is not case-sensitive.
VCOPY	Copies a range of lines from one variable and inserts them at a given line position in another variable.
VDELETE	Deletes a range of consecutive lines in a variable.
VFIND	Finds all lines containing occurrences of a specified string in a range of lines in a variable. VFIND is not case-sensitive.
VINSERT	Inserts lines from the TACL IN file at a given line position in a variable.
VLIST	Lists a range of consecutive lines in a variable.
VMOVE	Deletes a range of lines from one variable and inserts them at a given line position in another variable.

Unlike the built-in functions, the commands in Table 2-2 do not return a result. Instead, each of these commands (except VINSERT) lists the lines it operates on, with sequence numbers, to either the TACL OUT file or another user-specified file. You can append copies of the lines to an existing variable.

To use these commands, specify a range of line numbers or all of the lines in the variable (the default). If you use these commands to search for text or change text, the process is not case-sensitive; TACL performs the operation on all instances of the text. To specify a case-sensitive text change, use a line-editing or character-editing function, listed in Table 2-1.

The following code reserves the name of a variable called `sample2`:

```
12> #PUSH sample2
```

(The variable is initialized as soon as you store data in it.)

To insert several lines into the variable, starting at line 1, enter:

```
13> VINSERT sample2 1

1  The name of this variable is "sample2."
2  There are 37 characters in this line.
3  This is the last line in the variable.
```

To terminate input, type CTRL/Y (or a line that contains two slashes—//).

To display the contents of sample2, enter:

```
14> VLIST sample2

1  The name of this variable is "sample2."
2  There are 37 characters in this line.
3  This is the last line in the variable.
```

To find all occurrences of the word “line” in sample2, enter:

```
15> VFIND sample2 "line"

2  There are 37 characters in this line.
3  This is the last line in the variable.
```

To change all occurrences of “line” to “sentence,” enter:

```
16>  VCHANGE sample2 "line" "sentence"

2  There are 37 characters in this sentence.
3  This is the last sentence in the variable.
```

You can also use a variable for the comparison string:

```
17> #PUSH var1
18> #SET var1 line
19> VFIND sample2 var1

2  There are 37 characters in this line.
3  This is the last line in the variable.
```

Use the macro in Figure 2-6, `volnames`, to display the volume names on your system, using the VFIND global editing command. To run this macro, load the file and type `volnames`:

```
12> volnames

The volume names are:
$SYSTEM
$DATA
$DATA2
$DATA3
13>
```

This macro uses the OUTV construct to retrieve process output. For more information about OUTV, see "Using INV and OUTV" in Section 5, "Initiating and Communicating With Processes."

Figure 2-6. Retrieving Disk Names From DSAP

```
?SECTION volnames MACRO
#FRAME
[#PUSH
  dsapoutput         == an output stack for DSAP
  diskpaths          == a stack of disk names
  diskinfo           == a single line from DSAP
  volname            == a volume name
  displayinfo        == a stack of volume names to display
  count              == a stack counter
  begin              == beginning of the volume name
  end                == end of the volume name
]
== Run DSAP and capture the output:
DSAP /OUTV dsapoutput/ *, SHORT
VFIND /QUIET, TO diskpaths/ dsapoutput "$"

== Set a stack counter for diskpaths
#SET count [#LINECOUNT diskpaths]
==  Loop through the contents of diskpaths
[#LOOP |WHILE| [count] <> 0 |DO|
  ==  Extract a line and put it in diskinfo
  #SET diskinfo [#EXTRACT diskpaths]
  ==  Get values for the positions of the "$" and "-"
  ==  characters
  #SETMANY volname, [diskinfo]
  #APPENDV displayinfo volname
  #SET count [#COMPUTE [count] - 1]
]
#OUTPUT The volume names are:
#OUTPUTV displayinfo
#UNFRAME
```

Additional Data Manipulation Capabilities

Use the built-in functions and commands in Table 2-3 to perform other data manipulation tasks. A dash indicates that there is no equivalent function or command.

Table 2-3. Data Manipulation Functions

Function	Command	Description
#APPEND	—	Adds a line of text to a variable level.
#APPENDV	—	Appends a string or the contents of a variable level to the end of another variable level.
#COMPAREV	_COMPAREV	Compares one string or variable level with another.
—	COPYVAR	Copies the contents of one variable level to another.
#EMPTY	—	Determines whether specified text is empty.
#EMPTYV	—	Determines whether a variable level or quoted text is empty.
#EXTRACT	—	Obtains the first line of a variable level.
#EXTRACTV	—	Moves the first line of a variable level to another variable level.
—	FILETOVAR	Copies data from a file to the end of a variable level.
—	JOIN	Converts a multiple-line variable level into a single-line variable level with spaces in place of end-of-line indicators.
—	_LONGEST	Returns the longest element in a variable containing a space-separated list.
#SET	SET VARIABLE	Changes the entire contents of a variable level.
#SETMANY	—	Distributes the members of a space-separated list into individual variable levels.
#SETV	—	Copies a string or variable level into another variable level.
—	VARTOFILE	Copies data from a variable level to a file.

To read data into a variable from a sequential file, you can use either FILETOVAR or #SET. The following statements move the contents of *filename* into *variable*:

```
FILETOVAR filename variable
```

or

```
#SET /IN filename/ variable
```

The #SET command is much faster and is easier to debug: FILETOVAR is implemented as a looping macro; #SET uses TAL code. The maximum record length for records in files copied by FILETOVAR is 239 characters. When using #SET, make sure #INFORMAT is set to PLAIN.

To obtain input from a terminal, you can use #INPUTV, #SET, VINSERT, or #APPEND. The following function calls wait for input and store the input into *variable*:

```
#INPUTV variable prompt
#SET variable [#INPUT prompt-text]
#APPEND variable [#INPUT prompt-text]
```

In addition, you can use #REQUESTER to read from a terminal. For additional information about these built-in functions and commands, see the *TACL Reference Manual*.

Data Types

TACL does not support explicit data type definitions except in STRUCT variables. If you need to determine the data type of variable contents, such as whether the first constant in a variable is a number or a text constant, you can check to see if the type matches one of the alternatives supported by the #ARGUMENT built-in function. You can access a routine that contains an #ARGUMENT call from any type of procedural variable. For more information, see "Processing Arguments" in Section 3, "Developing TACL Routines."

Accessing Time Data

The system clock keeps track of time as a numeric value known as a timestamp. TACL supports timestamps in four formats for arithmetic operations, comparisons, and display purposes.

Timestamp Formats

The four timestamp formats differ in content and form:

- Julian timestamp, a four-word timestamp based on the Julian calendar. This timestamp represents the number of microseconds since 12:00 January 1, 4713 B.C, using Greenwich mean time (GMT). A GMT timestamp is stored as a four-word timestamp; for example:

  ```
  211479971400000000
  ```

  The Julian date includes a Julian day number—the integral number of days since January 1, 4713 B.C. The operating system assumes that the Julian day number starts at midnight local or Greenwich mean time, depending on the base of the timestamp.

- Local timestamp, a three-word timestamp. This timestamp represents the number of centiseconds (.01 second) since 00:00 December 31, 1974. An example is:

  ```
  45553140000
  ```

  A local timestamp can represent one of the following three time zones:

  - Local civil time (LCT): The time of day locally. This is in either standard time or daylight-saving time, depending on the area and the time of year.
  - Local standard time (LST): The time of day expressed in standard time.
  - Local daylight time (LDT): The time of day expressed in daylight-saving time. The daylight-saving time system extends the amount of daylight in the evenings by advancing the civil time. Usually, but not always, this is done in hour increments.

- Numeric date, a space-separated list in Gregorian date form—year, month, day, hour, minute, second, and fraction of a second. If obtained for a Julian timestamp, the list starts with the Julian day number. Examples include:
  - 1992 4 28 22 59 17 88—converted from a three-word timestamp
  - 2447635 1992 4 28 22 59 17 88—converted from a four-word timestamp
- Textual date, in Gregorian form, such as:

  ```
  May 7, 1992 08:30:00
  ```

Retrieving a Timestamp

Use the following built-in functions to retrieve date and time information from the system:

- #JULIANTIMESTAMP obtains the current timestamp, in Julian (four-word) format.
- #TIMESTAMP obtains the current timestamp, in local (three-word) format.

For example:

```
14> #TIMESTAMP

#TIMESTAMP expanded to:
54861863444

15> #JULIANTIMESTAMP

#JULIANTIMESTAMP expanded to:
211573083848182841
```

Converting a Timestamp

As noted previously, you can retrieve timestamps from the system in a three-word or four-word format. In addition, the #FILEINFO function returns timestamps in a three-word or four-word format, depending on the option you select. After you obtain the timestamp, you can include it in a function call, use it for calculations, or display it.

Use the functions and commands in Table 2-4 to convert between time and date representations. CONTIME is an abbreviation for converted time.

Table 2-4. Timestamp Conversion Functions

Starting Timestamp	Ending Timestamp	Function
Four-word (Julian)	Numeric date, including Julian day	#INTERPRETTIMESTAMP
Three-word	Numeric date	#CONTIME
Numeric date	Four-word	#COMPUTETIMESTAMP
Numeric date	Julian day number	#COMPUTEJULIANDAYNO
Three-word	Four-word	#CONVERTTIMESTAMP
Four-word	Three-word	#CONVERTTIMESTAMP
Julian day number	Numeric date	#INTERPRETJULIANDAYNO
Numeric date	Textual date	_CONTIME_TO_TEXT
Numeric date	Textual date without time	_CONTIME_TO_TEXT_DATE
Numeric date	Textual time	_CONTIME_TO_TEXT_TIME

In addition, the _MONTH3 function translates a two-digit month number to a three-letter month abbreviation.

Figure 2-7 shows the TACL functions that transform system timestamps from four-word and three-word formats to display format. Figure 2-8 shows how to convert timestamps returned by #FILEINFO into different formats.

Figure 2-7. Relationships Between System Timestamps and TACL Functions

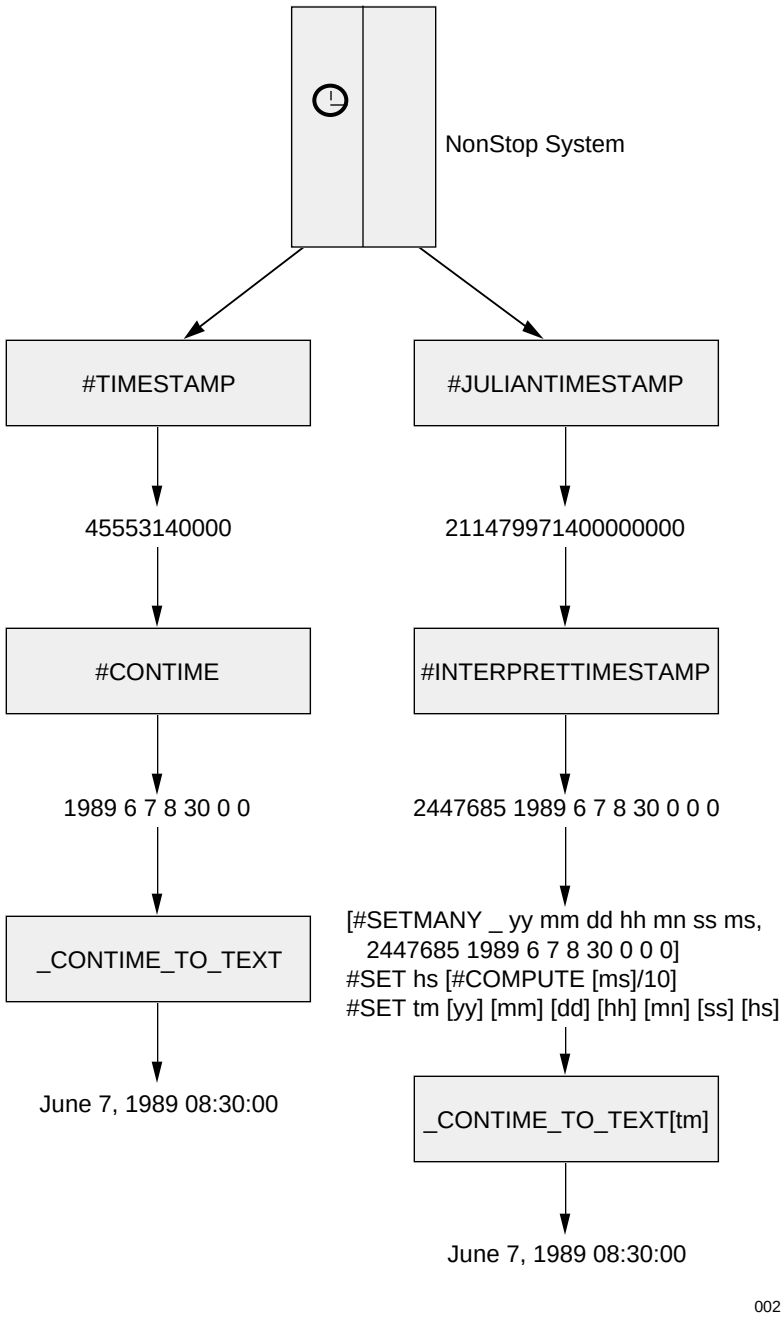
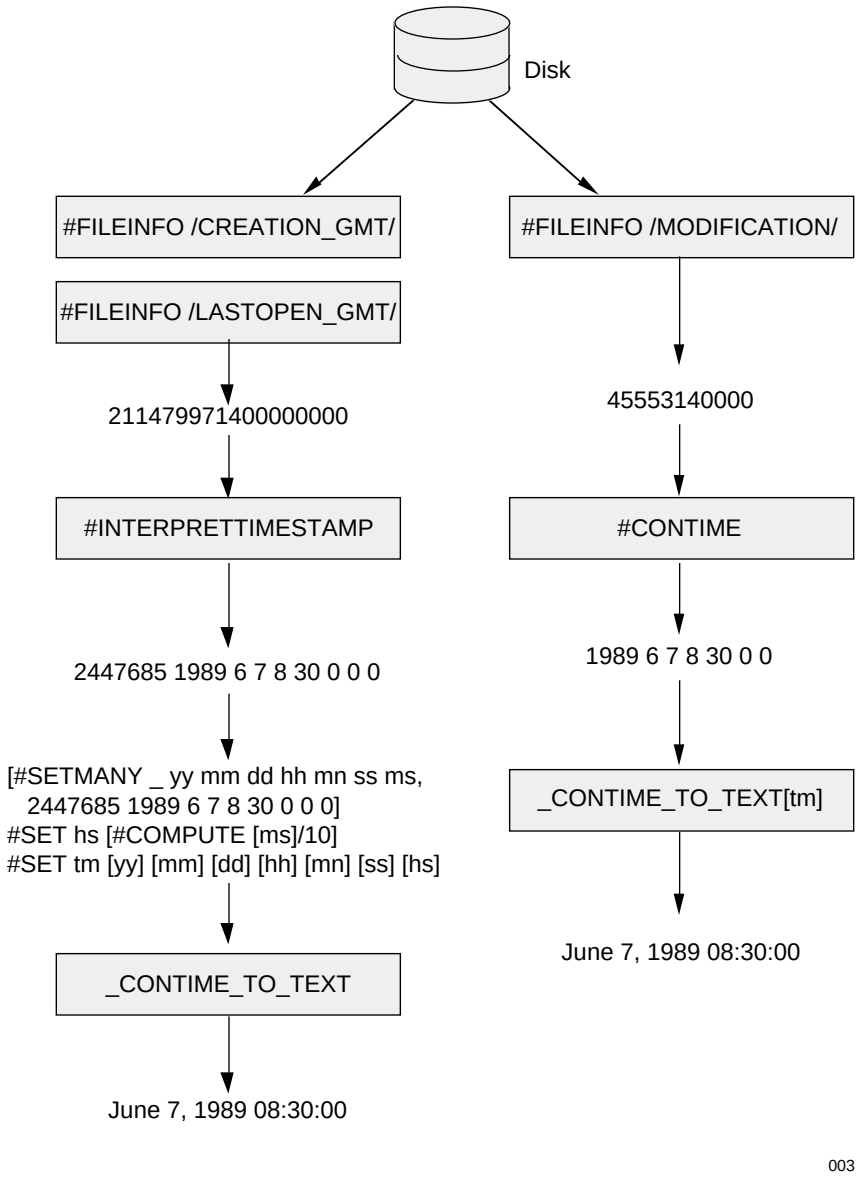


Figure 2-8. Relationships Between #FILEINFO Timestamps and TACL Functions



Computing the Day of the Week

Use the macro in Figure 2-9, dayofweek, to calculate the day of the week, as follows:

```
16> dayofweek
Tuesday
17>
```

Figure 2-9. Computing the Current Day

```
?SECTION dayofweek MACRO
== Calculates the day of the week from the three word
== TIMESTAMP, which starts at 00:00 31 December 1974
== (a Wednesday).

#FRAME
[#PUSH
  day^of^week
  elapsed^days
  elapsed^weeks
]

== There are 8640000 hundredths of a second in a day.
== Determine how many days have elapsed since
== midnight, December 31st, 1974 (starting timestamp).
#SET elapsed^days [#COMPUTE [#TIMESTAMP]/8640000]

== Determine how many whole weeks have elapsed.
#SET elapsed^weeks [#COMPUTE [elapsed^days]/7]

== The day of the week is the total elapsed days minus the
== number of days in the elapsed whole weeks.
#SET day^of^week [#COMPUTE [elapsed^days] -
  ([elapsed^weeks]*7)]

== Output the day
[#CASE [day^of^week]
  |0| #OUTPUT Tuesday
  |1| #OUTPUT Wednesday
  |2| #OUTPUT Thursday
  |3| #OUTPUT Friday
  |4| #OUTPUT Saturday
  |5| #OUTPUT Sunday
  |6| #OUTPUT Monday
  |OTHERWISE|
      #OUTPUT Error: Day of week must be 0-6 inclusive
]
#UNFRAME
```

Converting Timestamps Into Different Formats

The macro in Figure 2-10, `getdates`, converts the current date from a three-word timestamp to SQL format (yyyy-mm-dd), for use by report generators or other programs. This type of macro could be used to generate SQL reports.

Figure 3-9 in Section 3, "Developing TACL Routines," contains a modification of this example that shows how a nested routine can return dates as results.

Figure 2-10. Converting Timestamps

```
?SECTION getdates MACRO
#FRAME

== Save the current setting of #OUTFORMAT,
== so that it can be restored later:
#PUSH #OUTFORMAT #INFORMAT

[#PUSH
  date                == starting date (30 days ago)
  yyyy                == year
  mm                  == month
  dd                  == day
]

#SET #OUTFORMAT PRETTY

== Get the date and convert to yyyy mm dd calendar format:
#SETMANY yyyy mm dd, [#CONTIME [#TIMESTAMP]]

== Store as yyyy-mm-dd format (SQL date format):
#SET date [yyyy]-[mm]-[dd]

== Display the results to the user
#OUTPUT The reports will use a date of [date]
{ place report-generation code here}

#UNFRAME
```

Accessing Terminals

You can use TACL to read and write from a terminal. Sample operations include:

- ☐ Defining function keys
- ☐ Sending escape sequences to a terminal
- ☐ Changing the TACL prompt
- ☐ Implementing menus

In addition, you can write programs that execute one or more commands that you use frequently. The *Guardian User's Guide* describes how to create command definitions; these definitions are typically alias or macro variables.

Defining Function Keys

TACL recognizes 16 function keys. In the unshifted position they are named F1 through F15 (TACL predefines F16 as its help key). In the shifted position the function keys are named SF1 through SF16. You can define each function key to perform a sequence of operations that are useful in your environment. For example, you could specify F1 as TIME and F2 as FILES.

To define function keys, create an edit-format file that contains definitions for each function key you want to define. You then load the file (or refer to it from your TACLCSTM file so that it is loaded when you log on).

Function key definitions are typically ALIAS variables (if you are providing an alias for a single command or built-in function) or MACRO variables (for more complex operations). For example, you can redefine #LOGOFF with an alias, but for #LOGOFF/SEGRELEASE/ you must use a macro variable. The following code defines two alias variables:

```
?SECTION f1 ALIAS
TIME
?SECTION f2 ALIAS
FILES
```

You can, additionally, define characters or sequences of characters that perform a sequence of operations. For example, the following macro performs a FILENAMES operation on the specified subvolume (or on the current subvolume, if the user does not specify a subvolume):

```
?SECTION fn MACRO
FILENAMES %*%
```

To perform the FILENAMES operation, type FN.

The following macro prints a file to $S.#LP:

```
?SECTION pr MACRO
FUP COPY %*%, $S.#LP
```

To print a file, type PR *filename*.

For more information about defining function keys, see the *Guardian User's Guide*.

Sending Escape Sequences to a Terminal

The 6530 terminal and devices that emulate the 6530 terminal recognize a set of escape characters that allow you to control cursor position, set the size and video attributes of characters displayed on the screen, and perform other operations. Most escape sequences have the following format:

```
escape-character number [number]
```

The escape character is the 27th character in the ASCII character set. To specify an escape sequence, store binary values as their decimal equivalents in a STRUCT or DELTA variable. The easiest way to send these values to your terminal is to use #OUTPUT.

The following examples use STRUCT variables to store escape sequences. To specify an escape sequence in a STRUCT variable, set a BYTE value to the decimal value of the escape character and the numbers that perform the desired operation.

To define an escape sequence using #DELTA, specify a decimal number followed by the I command; for information about #DELTA, see the *TACL Reference Manual*.

Use the macro in Figure 2-11, `display`, to define several escape sequences and send them to the home terminal. To run this macro, load the file and type `display`. The macro displays a series of lines with different display attributes.

Figure 2-11. Sending Special Characters to a Screen (Page 1 of 2)

```
?SECTION display MACRO
#FRAME

== Define escape sequences
[#DEF ascii STRUCT
  BEGIN
    BYTE byt0 VALUE 7;
    CHAR bell REDEFINES byt0;
    BYTE byt1 VALUE 27;
    CHAR esc  REDEFINES byt1;
    BYTE byt2a VALUE 36;
    CHAR dollar REDEFINES byt2a;
    BYTE byt2 VALUE 37;
    CHAR perc REDEFINES byt2;
    BYTE byt3 VALUE 38;
    CHAR amp  REDEFINES byt3;
    BYTE byt4 VALUE 64;
    CHAR at   REDEFINES byt4;
    BYTE byt5 (0:1) VALUE 27 73;     == clear screen
    CHAR clr  (0:1) REDEFINES byt5;  == escape sequence
  END;

] == End ascii
```

Figure 2-11. Sending Special Characters to a Screen (Page 2 of 2)

```
== Clear the screen
#OUTPUT [ascii:clr(0:1)]
#OUTPUT The screen was just cleared.

== Display text with special video attributes
#OUTPUT Blinking text: [ascii:esc]6b These words are &
blinking[ascii:esc]6[ascii:at]

#OUTPUT Inverted text: [ascii:esc]6[ascii:dollar]These words&
 are inverted [ascii:esc]6[ascii:at]

#OUTPUT Inverted text: [ascii:esc]6[ascii:perc]These words &
are inverted and dim [ascii:esc]6[ascii:at]

#OUTPUT [ascii:esc]o[ascii:esc]6[ascii:amp]These words are &
 inverted blinking in line 25[ascii:esc]6[ascii:at]

#OUTPUT Here is the bell...[ascii:bell]
#OUTPUT And this is normal text.
#UNFRAME
```

Figure 5-10, in Section 5, “Initiating and Communicating With Processes,” contains a sample routine that writes to line 25 of a specified terminal.

Use the routine in Figure 2-12, `displayinfo`, to list a screen full of lines and prompt the user to continue. To use this routine, load the associated file and enter:

```
displayinfo
```

This routine does not display an entire screen of text, but shows how the prompt works.

Figure 2-12. Displaying a Screen of Text

```
?SECTION displayinfo ROUTINE
#FRAME

#PUSH help_input help_prompt
[#DEF ascii STRUCT
  BEGIN
    BYTE byt0 (0:1) VALUE 27;
    CHAR esc  (0:1)  REDEFINES byt0;
    BYTE byt1 (0:1) VALUE 84;
    CHAR rdf1 (0:1) REDEFINES byt1;   == roll down
    BYTE byt2 (0:1) VALUE 75;
    CHAR rdf2 (0:1) REDEFINES byt2;   == erase to end of line
  END;
]
#SET help_prompt PRESS CTRL/Y to exit or any other key to &
continue:~

#OUTPUT Sample list text...
#OUTPUT Line 2...
#OUTPUT Line 3...

#INPUTV help_input help_prompt

== The following code erases the prompt
== and resumes output on the line where the prompt was.
[#IF [#INPUTEOF] |THEN|
  #OUTPUT [ascii:esc][ascii:rdf1][ascii:esc][ascii:rdf2]&
[ascii:esc][ascii:rdf1]

|ELSE|
  #OUTPUT Display more text...
  #OUTPUT Line 2...
  #OUTPUT Line 3...
]

#UNFRAME
```

Use the macro in Figure 2-13, lock, to lock a terminal until the user types the password.

Figure 2-13. Locking a Terminal

```
?SECTION lock MACRO
#FRAME
#PUSH pw                      == password
#PUSH prompt                  == prompt variable
#PUSH rslt                    == result of #CHANGEUSER call
#PUSH userinfo                == result of USERS call
#PUSH line
#SET rslt 0

== Define the clear screen escape sequence
[#DEF ascii STRUCT                == Clear the screen
  BEGIN
    BYTE byt0 (0:1) VALUE 27 73;  == decimal escape-I
    CHAR clear (0:1) REDEFINES byt0;
  END;
]
#OUTPUT [ascii:clear(0:1)]

== Disable break mode
#SET #BREAKMODE DISABLE

== Obtain information about the current user
USERS /OUTV userinfo/
#EXTRACTV userinfo line
#EXTRACTV userinfo line
[#SET prompt Password for [#USERNAME
  [#CHARGET userinfo 22 FOR 8]]: ]

== Read a password (no echo) and attempt to log on:
[#LOOP |DO|
  #INPUTV /NOECHO/ pw prompt
  #SET rslt [#CHANGEUSER [#USERNAME [#CHARGET userinfo &
    22 FOR 8]] [pw]]
  [#IF NOT [rslt] |THEN| == An error occurred
    #OUTPUT Invalid password!
  ]
|UNTIL| rslt
]

== After a successful logon, enable break mode and exit the
== macro:
#SET #BREAKMODE ENABLE
#UNFRAME
```

Changing the TACL Prompt

The SETPROMPT command allows you to change the standard TACL prompt so that it includes the current volume or subvolume name.

To make additional modifications, define a variable called _PROMPTER that contains the definition of the prompt. Within the _PROMPTER variable, set the #PROMPT built-in variable to the desired prompt text. To cause TACL to invoke _PROMPTER prior to displaying a prompt, set the #PREFIX built-in variable to -1.

To save the previous prompt, push the #PREFIX built-in variable before you set #PREFIX to the new prompt text.

The following code displays the node name, volume, and subvolume in the TACL prompt:

```
#SET #PROMPT -1
[#DEF _PROMPTER TEXT |BODY| #SET #PREFIX [#DEFAULTS]]
```

When you are working on your local system, the prompt looks like this:

```
$DATA.SVOL 10>
```

When you have used the SYSTEM command or #SYSTEM built-in function to access another node, the prompt looks like this:

```
\RSYS.$DATA.SVOL 10>
```

To install the prompt whenever a user logs on, add a section to the user's macro definition file:

```
?SECTION _PROMPTER MACRO
#SET #PREFIX [#DEFAULTS]
```

Add the following line to the TACLCSTM file:

```
#SET #PROMPT -1
```

Alternatively, you could use the FILEINFO command to obtain the node name, volume, and subvolume.

For more information about #PROMPT and #PREFIX, see the *TACL Reference Manual*.

Implementing Menus

You can use TACL to define menus. A menu displays a screen and allows users to press function keys to access information and utilities.

Use the text variable in Figure 2-14, `menu`, to generate a menu. The user can press function keys to start applications and utilities. Note that the menu text could be displayed by individual #OUTPUT calls within the #LOOP function; setting `displayvar` to the display text avoids multiple #OUTPUT calls. To run this code, load the file that contains the code and type `menu`.

Figure 2-14. Displaying a Menu (Page 1 of 2)

```
?SECTION menu TEXT
#FRAME
[#PUSH prompt prompt1 prompt2 fkey temp temp2
  displayvar done
]
#PUSH #OUTFORMAT        == Save the current value

#SET #OUTFORMAT PRETTY
#SET done 0
#SET prompt  Please select a function key:~_~_
#SET prompt1 Type file name, followed by <return> :~_~_
#SET prompt2 Type printer name ($S.#AD is the default) :~_~_
[#APPENDV displayvar
"************************************************
~_
~_
~_
                    TANDEM APPLICATION MENU
~_
              F1  PSMail
              F2  TEDIT
              F3  TGAL Document Processor
              F4  Peruse
              F5  TFORM Document Processor
~_
~_
~_
~_
~_
~_
~_
          SF1     Exit to TACL
          SF16    Log off
~_
    Version III                                6/92
************************************************"
]
```

Figure 2-14. Displaying a Menu (Page 2 of 2)

```
[#LOOP |DO|
  #OUTPUTV displayvar
  SINK [#INPUT /FUNCTIONKEY fkey/ [prompt]]
  [#CASE [fkey]
  | F1 |  SINK [psmail]
  | F2 |  #INPUTV temp prompt1
          TEDIT [temp]
  | F3 |  #INPUTV temp prompt1
          #SET temp2
          #INPUTV temp2 prompt2
          [#IF [#EMPTYV /BLANK/ temp2] |THEN|
            #SET temp2 $S.#AD]
          SINK [TGAL /IN [temp], OUT [temp2], NOWAIT/]
  | F4 |  SINK [PERUSE]
  | F5 |  #INPUTV temp prompt1
          #SET temp2
          #INPUTV temp2 prompt2
          [#IF [#EMPTYV /BLANK/ temp2] |THEN|
            #SET temp2 $S.#ADMIN]
          SINK [TFORM /IN [temp], OUT [temp2], NOWAIT/]
  |SF1 |  #SET done 1
  |SF16|  #UNFRAME
          #LOGOFF
  | OTHERWISE |
    #OUTPUT ** Invalid selection. Select a valid function key
  ]
|UNTIL| [done]
]
#OUTPUT
#OUTPUT Exiting Application Menu...
#OUTPUT
#UNFRAME
```

Debugging TACL Programs

The TACL debugger shows how TACL interprets code. It provides step-by-step execution, examination of control flow, and examination and modification of variables. The debugger is a separate function, invoked by TACL upon request.

For information about how to debug #DELTA code, see "#DELTA Built-In Function" in the *TACL Reference Manual*.

Enabling the TACL Debugger

You can enable the debugger interactively or from within a macro or routine:

- ☐ At a TACL prompt, after loading a macro or routine variable, but before invoking it, type:

```
10> BREAK variable
```

- ☐ From within a macro or routine, type the following to enable the debugger:

```
#SET #TRACE -1
```

When you enable the debugger, TACL waits for an instruction before it performs its first expansion. At this point, you can set breakpoints and either resume execution or step through the code.

Debugger Commands

When tracing is on, the TACL trace facility invokes the _DEBUGGER function prior to invoking a variable. The debugger displays the current history number:

```
-nnn-
```

At this point, you can enter a command. If you enter a TACL command, the debugger passes the command to TACL for execution. If you enter a _DEBUGGER command, _DEBUGGER executes the command. Table 2-5 lists _DEBUGGER commands.

Table 2-5. _DEBUGGER Command Syntax

Command	Description
B[REAK] [*variable*]	Sets a breakpoint on the specified variable or variable level. If you omit *variable*, TACL lists all breakpoints. Whenever you invoke *variable*, TACL stops executing the code and waits for input from your debugging terminal.
C[LEAR] [*variable* | *]	Clears the breakpoint for the specified variable or variable level. If you specify an asterisk (*), TACL clears all breakpoints.
D[ISPLAY] *variable*	Displays the contents of the specified variable or variable level.
M[ODIFY] *variable*	Allows you to enter new contents for the specified variable or variable level.
R[ESUME]	Resumes execution until the next breakpoint or until TACL finishes executing code.
ST[EP]	Performs one expansion. To step through the function, press the RETURN key after each subsequent prompt.

To reenter the debugger after using STEP, set a breakpoint on a variable that will be invoked later in your program. Next, type RESUME to run your program until TACL encounters the breakpoint or finishes the program. Note that setting a variable (such as #SET x 123) is not an invocation of the variable; [x] is an invocation of the variable.

To end a debugging session, clear all breakpoints and type RESUME.

The following considerations apply to use of the debugger:

- The debugger is itself a TACL variable. Any inputs that are not debugger commands are assumed to be TACL commands, variables, or built-in functions. Commands such as #UNFRAME can influence the routine that is being debugged.
- Debug commands must reference declared variables. The TACL debugger displays each line before it is evaluated; therefore, a declaration (#PUSH) is in effect when the debugger is displaying a line that follows the #PUSH function.
- To set a breakpoint, specify a variable that will be invoked at a later point in the debugging session. The variable may be used by the function you will debug or by a later invocation from within the program, but it must be defined when you set the breakpoint.

Note You cannot set a breakpoint on a variable that is located in a read-only segment file such as TACLSEGF.

A Sample Debugging Session

Use the routine in Figure 2-15, `tedsave`, as sample code for the interactive debugging session described following Figure 2-15.

The routine in Figure 2-15 invokes TEDIT for a file supplied as the argument. The syntax for this routine is:

```
tedsave file-name
```

The example in Figure 2-15 includes the use of #ARGUMENT, which is described in Section 3, “Developing TACL Routines.” To perform the same work from a macro,

without accessibility to the #ARGUMENT built-in function, you would need to check that the argument is a valid file name.

Figure 2-15. Starting TEDIT From TACL

```
?SECTION tedsave ROUTINE
#FRAME
#PUSH editfile

== Retrieve the first argument and place it into editfile:
[#CASE [#ARGUMENT /VALUE editfile/ FILENAME /SYNTAX/
    OTHERWISE]
  |1|
    TEDIT [editfile]
    #OUTPUT DONE WITH [#FILEINFO /FULLNAME/ [editfile]]
  |2|
    #OUTPUT *** Error:  invalid filename ***
] == end #CASE

#UNFRAME
```

When you run `tedsave`, it displays output similar to the following:

```
>2 tedsave sect08
```

Control passes to TEDIT. After the user exits from TEDIT, the routine displays:

```
DONE WITH $VOL.SUBVOL.SECT08
```

The following paragraphs describe a sample debugging session. The `tedsave` routine is already loaded into memory:

```
3> BREAK tedsave
4> tedsave sect08
tedsave sect08
          ^
-BREAK-
```

On line 5, the user issues a STEP command and _DEBUGGER displays the next line of the routine. The user then presses RETURN to continue stepping through the routine; when the user steps through the routine in this way, the line number does not advance (lines that show nothing but a line number are terminated by RETURN):

```
-5-STEP
#FRAME
-TRACE-
-6-
#PUSH editfile
-TRACE-
-6-
[#CASE [#ARGUMENT /VALUE editfile/ FILENAME /SYNTAX/
        ^
```

```
    OTHERWISE ]
-TRACE-
-6-
[#CASE 1
 ^
-TRACE-
-6-
TEDIT [editfile]
       ^
-TRACE-
```

The user issues a DISPLAY command to see the contents of the variable EDITFILE and sets a breakpoint on that variable:

```
-6-DISPLAY editfile
sect08
-7-BREAK editfile
```

The RESUME command terminates the debug mode, and processing continues until the routine is ready to invoke EDITFILE, at which point the set breakpoint invokes _DEBUGGER again, which displays the word -BREAK- to show why it was invoked:

```
-8-RESUME
TEDIT editfile
-BREAK-
```

The user displays the contents of EDITFILE again, clears the breakpoint, and resumes normal processing:

```
-9-DISPLAY editfile
sect08
-10-CLEAR editfile
-11-RESUME
  Control passes to TEDIT.  After you exit,
  the routine displays:
DONE WITH $VOL.SUBVOL.SECT08
```

To modify a variable during a debugging session, use the MODIFY command. After entering the new value, press RETURN, then enter CTRL/Y to signify that there is no more input. At prompt 10, in the previous example, you could type the following to change the name of the edit file:

```
-10-MODIFY editfile
Input new contents of :EDITFILE.1; end with eof
-:EDITFILE.1-sect09
-:EDITFILE.1-EOF!
-11-D editfile
sect09
-12-
```

By using the MODIFY command, you can determine how a change affects the program. You can also force choices based on variable values without having to change the function and rerun it. For example, you can alter the contents of the text in a #CASE function to force TACL to take a path that you want to test.

(This page left intentionally blank)

3 Developing TACL Routines

TACL routines provide features that you cannot obtain from any other type of TACL variable. In a routine, you can:

- ☐ Use #ARGUMENT to check the syntax and validity of several types of arguments or to parse data within your program
- ☐ Use #RESULT to return a specific result (instead of an expansion of text)
- ☐ Use #ROUTINENAME to obtain the name of the active routine, for issuing recursive calls
- ☐ Use #RETURN to exit from any location in the routine
- ☐ Create an exception handler that processes events or errors

The following subsections describe how to use these features.

Processing Arguments

When you invoke a routine, you can include a list of arguments after the routine name. A routine does not, however, access these arguments in the same manner as macro arguments (%*n*%). Instead, in your routine, you specify an #ARGUMENT function with a list of argument alternatives. The #ARGUMENT function steps through the list and checks to see if the current argument matches a specified alternative. If the argument matches, #ARGUMENT returns an index to the alternative and optionally stores the argument in a variable for use within the routine.

The following statement checks to see if the next argument is a valid subvolume name (SUBVOL alternative) or system name (SYSTEMNAME alternative):

```
#SET num [#ARGUMENT /VALUE name/ SUBVOL SYSTEMNAME]
```

If the argument is a valid subvolume name, #ARGUMENT assigns 1 to num, indicating that the argument is a subvolume name, and stores the qualified argument in name. (The VALUE option affects how the #ARGUMENT built-in function stores the argument. For more information, see the *TACL Reference Manual*.)

The following examples show differences between argument processing in macros and routines. The programs support the following syntax:

```
process_argm file-name
process_argr file-name
```

To process the *file-name* argument from a macro:

```
?SECTION process_argm MACRO
== This macro does not check argument type or validity.
== Separate coding is required to validate the argument.
#FRAME
#PUSH fname

== Store the first argument in fname.
#SET fname %1%

#OUTPUT File name is [fname]
#UNFRAME
```

To process the *file-name* argument from a routine:

```
?SECTION process_argr ROUTINE
== This routine checks for correct file name syntax and
== existence of the named file.
#FRAME
#PUSH fname rslt

== Check to see if the first argument is a valid file name
== for an existing file.  If so, store it in fname.
#SET rslt [#ARGUMENT /VALUE fname/ FILENAME OTHERWISE]
[#IF rslt = 1 |THEN|   == valid filename
  #OUTPUT File name is [fname]
|ELSE|
  #OUTPUT *** Invalid filename ***
]
#UNFRAME
```

Note The OTHERWISE alternative allows you to handle invalid arguments within your program.

#The routine performs more error checking. If, the user does not supply an argument, or if the file does not exist, the routine returns an error. In contrast, the macro continues with an invalid file name.

The resulting argument text may be different between macros and routines. In the previous example, the macro outputs exactly what it was given:

```
39> process_argm thisfile
File name is thisfile
```

The routine, because of the VALUE option in the #ARGUMENT call, returns the fully qualified file name. For example:

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

Table 3-1 lists the built-in functions that support arguments to routines.

Table 3-1. Functions That Support Arguments

Function	Description
#ARGUMENT	Allows you to define a list of argument types. Compares each argument against these types. If an argument matches a specified type, #ARGUMENT returns a number that indicates the position of the argument type in your list of types. You can optionally specify a variable that will contain the contents of the argument.
#GETSCAN	Returns the number of characters that #ARGUMENT has processed, not including the routine name and the first character after the name.
#MORE	Determines whether an entire argument set has been processed.
#RESET	Sets the argument pointer to the beginning of the argument list.
#REST	Returns the number of unprocessed arguments.
#SETSCAN	Specifies the position at which the next #ARGUMENT function will resume processing arguments.

How #ARGUMENT Works

Use the #ARGUMENT built-in function to specify data types and, in some cases, entities as arguments. When invoked, #ARGUMENT steps through the list of supplied arguments.

#ARGUMENT Options

The #ARGUMENT built-in function supports the following options:

- PEEK processes an argument but keeps the internal argument pointer at the current argument.
- TEXT specifies a variable to contain an exact copy of the argument.
- VALUE specifies a variable to contain the TACL interpretation of the argument sequence. For example, FILENAME returns a fully-qualified file name, using defaults if the user did not specify all components of the file name.

You specify options within slashes (/) after #ARGUMENT.

#ARGUMENT Alternatives

Argument types are called alternatives. You specify alternatives after options and their associated slashes. Alternatives include:

- Contiguous characters (CHARACTERS), a string (STRING), or a number (NUMBER)
- Special characters, including "/" (SLASH), "(" (OPENPAREN), ")" (CLOSEPAREN), and "," (COMMA)
- Keywords defined in the routine (KEYWORD), such as TYPE or AGE

- File names (FILENAME), DEFINE names and attribute names (DEFINENAME and ATTRIBUTENAME), process names (PROCESSNAME), system names (SYSTEMNAME), and user names (USER)
- Subsystem IDs (SUBSYSTEM) and text for SPI and EMS tokens (TOKEN)

Some alternatives allow you to limit processing to syntax checking. For example, the FILENAME alternative looks for the name of an existing file. If, however, you specify FILENAME /SYNTAX/, the #ARGUMENT built-in function searches for a file name that is formatted correctly; it does not check for the existence of the file.

If an argument does not match any of the listed alternatives, a TACL error occurs unless you specify the OTHERWISE alternative. If you use OTHERWISE, an invalid argument does not produce a TACL error; your routine must retrieve and examine the invalid argument and determine an appropriate action.

Using #ARGUMENT

To define a fixed order for arguments, use a sequence of #ARGUMENT statements. The following statements search for a file name, followed by a slash, followed by a variable name of type text:

```
#PUSH fn var
SINK [#ARGUMENT /VALUE fn/ FILENAME/SYNTAX/]
SINK [#ARGUMENT SLASH]
SINK [#ARGUMENT /VALUE var/ VARIABLE /ALLOW TEXT/]
```

The VALUE options cause the #ARGUMENT built-in functions to store the actual argument in the specified variable. If there is an error, the program ends with an error; otherwise, the SINK calls suppress the results of the #ARGUMENT calls. (Each #ARGUMENT statement processes one type of argument, so the result is always 1 unless an error occurs.)

To process several types of arguments entered in any order, use a #CASE statement. Use the routine in Figure 3-1 to process zero or more of the following:

- File attribute names (defined in ALL)
- Numbers

Figure 3-1. Processing Arguments

```
?SECTION mult_args ROUTINE
#FRAME
#PUSH stat count end
#SET count 0
#SET end 0

[#DEF attributes TEXT |BODY|
  Type Size Age Owner Security
]

#OUTPUT Entered arguments were:  [#REST]
[#LOOP |DO|
  #SET count [#COMPUTE [count] + 1]
  [#CASE [#ARGUMENT/VALUE stat/
      KEYWORD/WORDLIST [attributes]/ NUMBER END OTHERWISE]
    |1|
      #OUTPUT Argument [count] is the keyword [stat].
    |2|
      #OUTPUT Argument [count] is the number [stat].
    |3|
      #SET end 1
    |4|
      #OUTPUT *** Invalid argument ***
      #SET end 1
  ]
|UNTIL| (NOT [#MORE]) OR end
]
#UNFRAME
```

Using #ARGUMENT for Data Within a Program

TACL itself does not provide data type declarations and functions, but you can write a function that uses #ARGUMENT to determine the type and return the information. The following routine returns TRUE (not zero) if you pass it a number; otherwise, it returns FALSE:

```
?SECTION anumber ROUTINE
  #RESULT [#COMPUTE NOT ([#ARGUMENT NUMBER OTHERWISE] - 1)]
```

Variations on this routine could return TRUE for text or special characters, a number that reflects a group of argument types, or an index into the entire set of #ARGUMENT alternatives.

The following examples illustrate two ways to retrieve a number from a position within a line of text (as returned by FUP or other processes). First, you can use the #CHARGET function:

```
#PUSH pfree line
#SET line This is a test number: 53
#SET pfree [#CHARGET line 24 FOR 2]
#OUTPUT [pfree]
```

The preceding code retrieves two characters from `line`, but does not check that the two characters are numbers. The number at position 24 must be two characters long; the code returns two digits even if the number has a single digit or three digits.

As an alternative, you can define a routine that uses #ARGUMENT.

```
?SECTION getnumber ROUTINE
#FRAME
#PUSH rslt arg position

== First, get the requested character position
#SET rslt [#ARGUMENT /TEXT position/ NUMBER OTHERWISE]
[#CASE [rslt]
  |1|
    == position is OK
  |2|
    == Caller did not supply a number for position arg.
    #RESULT -1
    #RETURN
]
== Skip [position] characters
SINK [#ARGUMENT CHARACTERS /WIDTH [position]/ ]
#SET rslt [#ARGUMENT /TEXT arg/ NUMBER OTHERWISE]

[#CASE [rslt]
  |1|
    #RESULT 0 [arg]
  |2|
    == Invalid argument; text at specified position is
    == not a number
    #RESULT -2
]
#UNFRAME
```

If `getnumber` finds a number, it returns a zero followed by the requested number. If `position` is invalid, it returns -1; otherwise, it returns -2.

You could call this routine from your program:

```
?SECTION caller ROUTINE
#FRAME
== call getnumber with the starting position
== and the line of text

#PUSH pfree position line
#SET line This is a test number: 53
#SET position 23

#SET pfree [getnumber [position] [line]]
#OUTPUT [pfree]
#UNFRAME
```

The routine performs more checking than #CHARGET and is more flexible with the length of a space-separated number. The routine, however, takes longer to write, making it most useful if you plan to perform this action many times in your program.

Processing File Name Arguments

The following routine parses an OUT option enclosed in slashes. The routine first checks for a slash (/). If present, the routine checks for the word OUT, followed by a file name and an ending slash. The routine then sets the TACL OUT file to the file specified in the argument list. If the user does not specify an OUT option, this routine displays the current setting of the OUT file. To run the routine, type the name of the file that contains the code:

```
?TACL ROUTINE
#FRAME
#PUSH out outfile
[#CASE [#ARGUMENT SLASH END OTHERWISE]
  |1|               == Found the first slash character
    SINK [#ARGUMENT KEYWORD /WORDLIST out/]
    SINK [#ARGUMENT /VALUE outfile/ FILENAME /SYNTAX/]
    #OUTPUT [outfile]
    SINK [#ARGUMENT SLASH]
    SINK [#ARGUMENT END]
    #PUSH #OUT
    #SET #OUT [outfile]
  |2| == No arguments; display the current OUT file
    #OUTPUT The current OUT file is [#OUT]
  |OTHERWISE|         == Unknown argument
    #OUTPUT Invalid argument
]
#OUTPUT [outfile]
#UNFRAME
```

The following routine expects both a file name and a properly formatted variable name, but accepts them in either sequence. To run this routine, type the name of the file that contains this code:

```
?TACL ROUTINE
#FRAME
#PUSH fname vname
[#CASE [#ARGUMENT /VALUE fname/ FILENAME VARIABLE /SYNTAX/]
  |1| #IF [#ARGUMENT /VALUE vname/ VARIABLE /SYNTAX/]
  |2| #SETV vname fname == fname contains the variable name;
                         == move the variable name into vname.
      #IF [#ARGUMENT /VALUE fname/ FILENAME]
]
#OUTPUT vname = [vname]
#OUTPUT fname = [fname]
#UNFRAME
```

Processing Variables as Arguments

The following routine accepts any type of existing variable except a STRUCT or a STRUCT item:

```
?TACL ROUTINE
#PUSH vname
#IF [#ARGUMENT /VALUE vname/ VARIABLE /FORBID STRUCT ITEM/]
#OUTPUT [vname]
#POP vname
```

Processing a Space-Separated List of Words

The following routine accepts a space-separated list of words and returns it as a comma-separated list.

```
?SECTION wordlist ROUTINE
#FRAME
#PUSH wurd
[#LOOP |WHILE| [#MORE] |DO|
  #IF [#ARGUMENT /VALUE wurd/ WORD /SPACE/]
  #RESULT [wurd][#IF [#MORE] |THEN|,]
]
#UNFRAME
```

If you run `wordlist` interactively, load the file that defines `wordlist` and use #OUTPUT(V) to display the result:

```
15> #OUTPUT [wordlist a b c]
a, b, c
16>
```

For information about #RESULT, see “Returning Results,” later in this section.

Processing Arguments Recursively

You can use the #ROUTINENAME built-in function to process arguments recursively. For more information, see "Calling a Routine Recursively," later in this section.

Examining the Contents of Arguments

The following routines examine the contents of arguments. These routines return results; if you use the routines interactively, use #OUTPUT(V) to display results.

Use the routine in Figure 3-2, `first`, to retrieve a specified number of characters in a variable. The syntax is:

```
first variable number
```

Figure 3-2. Returning Characters From a Routine

```
?SECTION first ROUTINE
#FRAME
#PUSH var num

#IF [#ARGUMENT/VALUE var/VARIABLE]
#IF [#ARGUMENT/VALUE num/NUMBER]
#IF [#ARGUMENT END]

== Enclose the following in brackets, in case the result
== contains more than one line
[#RESULT [#CHARGET [var] 1 FOR [num]]]

#UNFRAME
```

To obtain the contents of a multiple-line variable, enclose your statement in square brackets. If, for example, `x` contains the following:

```
abcde
fgh
```

you can display the contents of `x` (including the line break character at the end of the first line) with the following statement:

```
15> [#OUTPUT [first x 9]]
abcde
fgh
16>
```

The result includes an end-of-line character. For information about #RESULT, see "Returning Results," later in this section.

Use the routine in Figure 3-3, `substring`, to retrieve characters from position *number1* to position *number2* of a variable. The syntax is:

```
substring variable number1 number2
```

In this and the following three examples, the #RESULT function call is enclosed in brackets in case the #CHARGET built-in function returns more than one line.

Figure 3-3. Returning a Set of Characters From a Variable

```
?SECTION substring ROUTINE
#FRAME
#PUSH bgn end var

#IF [#ARGUMENT/VALUE var/VARIABLE]
#IF [#ARGUMENT/VALUE bgn/NUMBER]
#IF [#ARGUMENT/VALUE end/NUMBER]
#IF [#ARGUMENT END]

[#RESULT [#CHARGET [var] [bgn] TO [end]]]
#UNFRAME
```

Note This routine does not check to make sure that *number2* is a greater number than *number1*. For more thorough argument validation, include that check.

Use the routine in Figure 3-4, scan, to scan for text and retrieve the first position, starting at a specified position, where the text occurs in the variable. The syntax is:

```
scan variable number text
```

Do not enclose *text* in double quotes unless the quotes are part of the text.

Figure 3-4. Searching for Text

```
?SECTION scan ROUTINE
#FRAME
#PUSH var num txt

#IF [#ARGUMENT/VALUE var/VARIABLE]
#IF [#ARGUMENT/VALUE num/NUMBER]
#IF [#ARGUMENT/VALUE txt/TEXT]
#IF [#ARGUMENT END]

[#RESULT [#CHARFINDV [var] [num] txt]]
#UNFRAME
```

Note If you do not include square brackets around the variable names in the #CHARxxx calls, TACL uses the declared variables (var and num) instead of the variables passed to the routine and referenced by var and num.

Use the routine in Figure 3-5, length, to retrieve the number of characters in a variable (including end-of-line characters). The syntax is:

```
length variable
```

Figure 3-5. Counting Characters in a Variable

```
?SECTION length ROUTINE
#FRAME
#PUSH var

#IF [#ARGUMENT/VALUE var/VARIABLE]
#IF [#ARGUMENT END]

[#RESULT [#CHARCOUNT [var]]]
#UNFRAME
```

Use the routine in Figure 3-6, `insert`, to insert the contents of *`variable1`* into *`variable2`* at the specified position. The syntax is:

```
insert variable1 variable2 number
```

Figure 3-6. Moving Text Between Variables

```
?SECTION insert ROUTINE
#FRAME
#PUSH var1 var2 num

#IF [#ARGUMENT/VALUE var1/VARIABLE]
#IF [#ARGUMENT/VALUE var2/VARIABLE]
#IF [#ARGUMENT/VALUE num/NUMBER]
#IF [#ARGUMENT END]

[#RESULT [#CHARINSV [var2] [num] [var1] ]]
#UNFRAME
```

Parsing Arguments for a Caller

You can use the #ARGUMENT built-in function to provide a general parser for other TACL programs. Figure 3-7 contains two sample programs:

- `getargs`, a macro that parses arguments for a routine that calls it and returns the value in a variable (the calling program supplies a name)
- `call_getargs`, a routine that calls `getargs`

Note `Getargs` is defined as a macro, and can be used only when called by a routine. Otherwise, the #ARGUMENT call inside the macro is not valid.

The syntax for `getargs` is:

```
getargs triplet [ triplet ]...
```

where *`triplet`* contains three parts:

```
{ REQUIRED } type variable
{ OPTIONAL }
```

- REQUIRED or OPTIONAL specifies whether the corresponding argument is required or optional.
- *`Type`* is an #ARGUMENT alternative such as FILENAME. The specified type must not permit spaces. The KEYWORD alternative, for example, cannot be used because it requires the WORDLIST alternative, which allows spaces in its syntax.
- *`Variable`* is a name for the argument. `Getargs` stores the argument value in a variable with this name. `Getargs` pushes *`variable`* and sets it if a matching argument is found; otherwise, *`variable`* is empty.

The following statement asks `getargs` to search for two arguments—one required number and one optional text constant:

```
getargs REQUIRED NUMBER numvar OPTIONAL TEXT datavar
```

If `getargs` finds a numeric argument, it pushes `num` and stores the argument into `num`. If `getargs` finds a text argument, it pushes `data` and stores the argument in `data`.

Figure 3-7. Assigning Values to Arguments (Page 1 of 2)

```
?SECTION getargs MACRO

== Loop through all triplets
[#IF [#EMPTY %1%] |THEN|
  == No more triplets; routine must have no more arguments
  SINK [#ARGUMENT END]   == Only valid if called by a routine
|ELSE|
  #PUSH %3%              == Push the variable
  == Check first word of triplet for REQUIRED or OPTIONAL
  [#CASE %1%
    |optional|
      [#IF [#EMPTY %4%] |THEN|  == Check for more triplets
        == No more triplets; argument cannot be followed by
        == a comma
        [#CASE [#ARGUMENT/TEXT %3%/ %2% END]
          |1|
            SINK [#ARGUMENT END]
          |2|
            #SET %3%
          |3|
        ]
```

Figure 3-7. Assigning Values to Arguments (Page 2 of 2)

```
      |ELSE|
        == More triplets; argument can be followed by a comma
        [#CASE [#ARGUMENT/TEXT %3%/ %2% COMMA END]
          |1|
            SINK [#ARGUMENT COMMA END]
          |2|
            #SET %3%
          |3|
        ]
    ] == end #IF

    |required|
      SINK [#ARGUMENT/TEXT %3%/ %2%] == Get required argument
      [#IF [#EMPTY %4%] |THEN|     == Check for more triplets
        == No more triplets; argument cannot be followed by
        == a comma
        SINK [#ARGUMENT END]
      |ELSE|
        == More triplets; argument can be followed by a comma
        SINK [#ARGUMENT COMMA END]
      ]  == end #IF
  ]  == end #CASE
  == Call self again, without the current triplet.
  %0% %4 TO *%
]  == end #IF
```

Call_getargs supports the following syntax:

```
call_getargs filename, filename [, number, number ]
```

The following shows a sample invocation of `call_getargs`:

```
11> call_getargs data1, data2, 4
file1 = data1
file2 = data2
n1 = 4
12>
```

Figure 3-8. Sending Arguments to a Parsing Program

```
?SECTION call_getargs ROUTINE
#FRAME
[getargs
  REQUIRED filename file1
  REQUIRED filename file2
  OPTIONAL number n1
  OPTIONAL number n2
]

== Display the results
#OUTPUT file1 = [file1]
#OUTPUT file2 = [file2]
[#IF NOT [#EMPTY [n1]] |THEN|
  #OUTPUT n1 = [n1]
]
[#IF NOT [#EMPTY [n2]] |THEN|
  #OUTPUT n1 = [n2]
]

#UNFRAME
```

Returning Results

Function results come from one or more #RESULT built-in functions within the routine. This is an important distinction between macros and routines: a macro invocation returns the expansion of the text of the macro; a routine returns only what the #RESULT function provides.

Use the macro in Figure 3-9, `report_shell`, to calculate today's date and the date thirty days ago and convert the dates to SQL format (yyyy-mm-dd). This example is similar to Figure 2-10, but returns results. To run this macro, load the associated file and enter:

```
report_shell
```

Figure 3-9. Converting Timestamps

```
?SECTION report_shell MACRO
#FRAME
#PUSH #OUTFORMAT #INFORMAT today
#SET #OUTFORMAT PRETTY
#SET #INFORMAT TACL
== Calling part of Macro

#OUTPUT This macro displays a start and end date.
#SETMANY today, [get_dates]
#OUTPUT
#OUTPUT The date is [today]
#UNFRAME

?SECTION get_dates ROUTINE
#FRAME

== Save the current setting of #OUTFORMAT,
== so that it can be restored later:
#PUSH #OUTFORMAT #INFORMAT

[#PUSH
  date                == starting date (30 days ago)
  yyyy                == year
  mm                  == month
  dd                  == day
]
#SET #OUTFORMAT PRETTY

== Get the date and convert to yyyy mm dd calendar format:
#SETMANY yyyy mm dd, [#CONTIME [#TIMESTAMP]]
== Store as yyyy-mm-dd format (SQL date format):
#SET date [yyyy]-[mm]-[dd]
== Return the result to the caller.
#RESULT [date]
#UNFRAME
```

Calling a Routine Recursively

#The #ROUTINENAME built-in function returns the name of the currently active routine, which allows you to invoke a routine from within the routine. The function is similar to accessing %0% from a macro, but you cannot use #ROUTINENAME in a macro or %0% in a routine.

If you call #ROUTINENAME for a routine defined with a ?TACL ROUTINE directive, #ROUTINENAME returns the name of the variable TACL uses to hold the active copy of the routine.

Use the macro in Figure 3-10, `caller`, to process one or more arguments. To use this macro, load the associated file and enter:

```
caller { file-name | TACL | TAL | PASCAL | system-name }...
```

`Caller` calls `proc_arg` to process each argument. `Proc_arg` calls itself additional times if there is more than one argument.

Figure 3-10. Processing Arguments

```
?SECTION caller MACRO
#FRAME
#PUSH e1 var1 rslt
[#IF NOT [#EMPTY %1%] |THEN|
  proc_arg %*%
]
#UNFRAME

?SECTION proc_arg ROUTINE
#SET rslt [#ARGUMENT /VALUE var1/ FILENAME KEYWORD  &
    /WORDLIST tacl tal pascal/ SYSTEMNAME OTHERWISE]

[#CASE [rslt]
  |1| == File name
    FILEINFO [var1]
  |2| == Keyword
    #OUTPUT [#SHIFTSTRING /UP / [var1]] is a keyword
  |3| == System name
    #OUTPUT /HOLD/ The system number for [var1] is :
    #OUTPUT [#SYSTEMNUMBER [var1]]
  |4| == Word
    #OUTPUT Expecting a file name, the word TACL, TAL,
    #OUTPUT or PASCAL, or a system name.
    #RETURN
]
[#IF [#MORE] |THEN|
  #OUTPUT                          == blank line
  [#ROUTINENAME] [#REST]
]
```

Use the routine in Figure 3-11, `argrec`, to process one or more file names, checking for syntax but not for file existence. File names can be separated by spaces or commas. After processing each file name, the routine scans ahead and skips over commas. It calls itself to process each additional file name.

Figure 3-11. Processing File Name Arguments

```
?SECTION argrec ROUTINE
#FRAME
#PUSH fn nextfn rslt
#SET rslt [#ARGUMENT /TEXT fn/ FILENAME /SYNTAX/ OTHERWISE]
[#IF [rslt] = 2 |THEN|
  #OUTPUT *** Invalid filename ***
  #RETURN
]
== Process the first argument
#OUTPUT current argument = [fn]

== Check for another file name
#SET rslt [#ARGUMENT /TEXT nextfn/ FILENAME /SYNTAX/ &
   COMMA END OTHERWISE]

[#CASE [rslt]
  |1|
  |2 3|
    == Next argument is a comma or end; ignore it
    #SET nextfn
  |OTHERWISE|
    #OUTPUT *** Invalid argument ***
    #RETURN
]

== If nextfn contains a file name or there are additional
== unprocessed arguments, call self, appending the results.
[#IF NOT [#EMPTY [nextfn] [#REST]] |THEN|
  #RESULT [#ROUTINENAME] [nextfn] [#REST]
]
#UNFRAME
```

Exiting From a Routine

To exit from a routine, use the #RETURN built-in function. #RETURN exits immediately and does not reset any frames unless you specify #UNFRAME or #RESET FRAMES prior to #RETURN.

You can use #RETURN to define several exit points within a routine. For examples of the use of #RETURN, see the next subsection, “Writing an Exception Handler.”

Writing an Exception Handler

An exception is an event or condition that requires special handling. If, for example, a user presses the BREAK key or enters alphabetic data when a number is expected, an exception occurs. TACL cannot detect a modem disconnect, but can detect and process other exceptions, including ones you define. TACL recognizes three types of exceptions:

- ☐ Pressing the BREAK key
- ☐ A TACL error, as defined in Section 2, “Developing TACL Programs.”
- ☐ A user-defined exception, such as an end-of-file, for which special handling may be necessary.

Any one of these exceptions causes TACL to search for an exception handler. An exception handler is a portion of code that performs actions after an exception. For example, if a TACL routine opens one or more files and then purges them when finished, the user could press the BREAK key while the files are still open. TACL would then close any open files, but would not purge them. If the routine contained an exception handler, TACL could close the files before exiting. Activities of exception handlers can include:

- ☐ Issuing error messages
- ☐ Resetting data defaults
- ☐ Terminating open INLINE processes
- ☐ Resetting frames or accumulated results
- ☐ Purging scratch files
- ☐ Performing INITTERM operations
- ☐ Passing information to the calling routine
- ☐ Returning control to the calling routine

If you declare local variables within the body of your routine, determine whether or not to delete these variables within the exception handler.

If an exception occurs and the current routine has no exception handler, TACL exits from the routine and returns control to the calling routine. TACL continues to backtrack through the chain of calling routines, exiting routines as it goes, until it finds a routine that can process the type of exception that occurred. TACL then reinvokes that routine to process the exception. If TACL finds no such routine, it performs its own exception handling—it resets frames and results and, if the exception is of type _ERROR, displays an error message. Similarly, you can nest routines that contain exception handlers. TACL uses the first exception handler that can process the type of exception that occurred.

Exception handlers provide a way to release control and deallocate resources. In addition, you can write an exception handler that does not permit a user to exit the routine. In this manner, you can write command shells that define a set of commands available to users.

Types of Exception Handlers

Exception handlers can be divided into two types, depending on how they return control:

- Release handlers that relinquish control to the calling procedure
- Keep handlers that retain control regardless of exceptions (usually for security purposes)

A routine can contain both types of exception handlers.

Constructing an Exception Handler

Use the built-in functions in Table 3-2 to construct exception handlers.

Table 3-2. Functions That Support Exception Handlers

Function	Description
#ERRORNUMBERS	Returns the most recent TACL error.
#ERRORTEXT	Intercepts error text that would have been written to the OUT file if there had been no exception handler.
#EXCEPTION	Returns the type of exception that invoked the exception handler: ☐ _CALL if the routine containing #EXCEPTION was invoked normally. ☐ The name of the exception if the routine was invoked in response to an exception that was listed in a #FILTER function.
#FILTER	Specifies the types of exceptions a routine can handle.
#RAISE	Causes an exception to occur.
#RETURN	Returns immediately from the routine.

As shown in Figure 3-12, a routine that contains an exception handler has the following structure:

- It begins with a #CASE statement immediately after the ?SECTION directive. This #CASE statement uses the #EXCEPTION built-in function to determine which exception occurred. The #CASE statement includes the following:
  - The first label in the #CASE is _CALL. This exception occurs as part of normal processing when a calling program invokes the routine.
  - Remaining labels identify the exceptions for which the handler can be invoked. Each case contains statements that handle the associated type of exception.

  Do not place a #FRAME call before the #CASE statement.
- The body of code for the routine follows the #CASE statement. This code includes a #FILTER call that lists the exceptions for which this code is protected—and which are defined as labels in the #CASE statement. You can change the setting of #FILTER as necessary during processing to enable or disable processing of specific exceptions.
- The routine ends with an #UNFRAME function.

Figure 3-12. Sample Release Handler Template

```
?SECTION name ROUTINE

== Exception handler ==
[#CASE [#EXCEPTION]
  |_CALL |
    == No action required when first called
  |_BREAK|
     == Code to handle BREAK goes here
  |_ERROR|
     == Code to handle errors goes here
]  == End CASE

== Beginning of body of routine ==

== Filter these exceptions:
#FILTER _BREAK _ERROR

== Body of executable code goes here

#UNFRAME
```

Keep and release handlers have slightly different structures, as shown in Table 3-3.

Table 3-3. Differences Between Keep and Release Exception Handlers

Keep Handler Contents	Release Handler Contents
The #CASE statement:	The #CASE statement:
Contains a #FRAME and variable declarations in the _CALL portion	Does not contain a #FRAME
Pushes global variables	Does not push global variables
Does not invoke #RESET	Invokes #RESET and #RETURN
The body of code ends with #UNFRAME	The body of code starts with #FRAME and ends with #UNFRAME

Creating a Release Exception Handler

A release exception handler processes exceptions and returns to the calling procedure. The _CALL path, taken when the routine is invoked by a calling program, typically requires no action.

Use the routine in Figure 3-13, `command_processor`, as a sample release handler. The routine requests commands from the user and allows the user to enter an ADD or SUB command. The routine then displays the command.

The body of the routine begins with a #FRAME function call and #PUSH (or PUSH) and #DEF entries to define variables, followed by a #FILTER function call that declares the exceptions against which the code that follows is to be protected.

If the user presses the BREAK key while the processing loop is running, TACL raises the _BREAK exception and reinvokes the routine. The #CASE function executes the _BREAK case, displays a message, and exits. If the user enters anything other than ADD or SUB, the #CASE in the loop raises _ERROR, and TACL reinvokes the routine; in this situation, the #CASE function takes the _ERROR path, displays a message, and exits. To invoke this routine, load the associated file and enter:

```
command_processor
```

Figure 3-13. Sample Release Handler

```
?SECTION command_processor ROUTINE

== Exception handler ==
[#CASE [#EXCEPTION]
  |_CALL |
    == No action required when first called
  |_BREAK|
    #OUTPUT BREAK key pressed.
    #RESET RESULTS FRAMES
    #RETURN
  |_ERROR|
    #OUTPUT Input error occurred.
    #RESET RESULTS FRAMES
    #RETURN
] == End CASE

== Beginning of body of routine ==
#FRAME
#PUSH cmd

== Filter predefined exceptions only
#FILTER _BREAK _ERROR

== Processing loop:  runs until invalid command or BREAK key
[#LOOP |DO|
  #SET cmd [#INPUT Enter cmd: ]  == Get value from terminal
  [#CASE [cmd]
    |ADD|
      #OUTPUT ADD
    |SUB|
      #OUTPUT SUB
    |OTHERWISE|
      #OUTPUT Invalid command
      #RAISE _ERROR
  ]  == End CASE
|UNTIL| 0 = 1  == (do forever)
]  == End LOOP

#UNFRAME
```

When you invoke `command_processor`, the output looks like this:

```
16> command_processor
Enter cmd: add
ADD
Enter cmd: clr
Invalid command
TACL error occurred.

17> command_processor
Enter cmd: <BREAK>
BREAK key pressed.
18>
```

Use the routine in Figure 3-14, `purgefiles`, to purge files based on file name templates. `Purgefiles` illustrates the use of #FILTER, #FILENAMES, and the TEMPLATES alternative for the #ARGUMENT built-in function. To use this routine, load the associated file and enter:

```
purgefiles [ ! ] file-template [ , file-template ] ...
```

The ! specifies purge without confirmation; without it, the routine prompts for each file. If the routine encounters a _BREAK exception, it displays a message with the number of files purged and the number not purged, and then exits.

Figure 3-14. Returning Information From a Release Handler (Page 1 of 3)

```
?SECTION purgefiles ROUTINE
[#CASE [#EXCEPTION]
|_CALL|
|_BREAK _ERROR|
  #PUSH errtext
  #ERRORTEXT /CAPTURE errtext/
  #OUTPUT Break or error terminated function.
  #OUTPUT
  [#IF NOT [#EMPTYV /BLANK/ errtext] |THEN|
    #OUTPUTV errtext
    #OUTPUT
  ]
  #OUTPUT Number of files purged = [filespurged]
  #OUTPUT Number of files not purged = [filesnotpurged]
  #UNFRAME
  #RETURN
]
#FRAME

== Filter _BREAK and TACL errors
#FILTER _BREAK _ERROR
[#PUSH filetemplate exclude prevname purgeerr opt firsttime
  filespurged filesnotpurged
]
```

Figure 3-14. Returning Information From a Release Handler (Page 2 of 3)

```
[#DEF wanttopurge ROUTINE |BODY|
  #RESULT -1
  [#IF [exclude] |THEN|
    #RETURN
  ]
  [#IF [#MATCH y [#INPUT Purge [prevname] (Y/N)?]] |THEN|
    #RETURN
  ]
  #RESET results
  #RESULT 0
] == end #DEF wanttopurge
[#DEF handletemplate MACRO |BODY|
  #SET firsttime 0
  [#IF ([#ARGUMENT SLASH OTHERWISE] = 1) |THEN|
    [#IF [#ARGUMENT KEYWORD /WORDLIST start/]]
    [#IF [#ARGUMENT /VALUE prevname/ FILENAME /SYNTAX/]]
    [#IF [#ARGUMENT SLASH]]
    #SET prevname [#FILEINFO /FULLNAME/ [prevname]]
    #SET firsttime -1
  ]
  [#LOOP |DO|
    [#IF [firsttime] |THEN|
      #SET firsttime [#FILEINFO /EXISTENCE/ [prevname]]
    ]
    [#IF [firsttime] |THEN|
      #SET firsttime 0
    |ELSE|
       #SET prevname [#FILENAMES /MAXIMUM 1, &
         PREVIOUS [prevname]/ [filetemplate]]
    ]
    [#IF NOT [#EMPTYV /BLANK/ prevname] |THEN|
      [#IF [wanttopurge] |THEN|
        #SET purgeerr [#PURGE [prevname]]
        [#IF [purgeerr] |THEN|
          #OUTPUT Purge error [purgeerr] on [prevname]
          #SET filesnotpurged [#compute filesnotpurged+1]
        |ELSE|
          #OUTPUT [prevname] purged
          #SET filespurged [#COMPUTE filespurged + 1]
        ]
      |ELSE|
        #SET filesnotpurged [#COMPUTE filesnotpurged + 1]
      ]
    ]
  |UNTIL| [#EMPTYV /BLANK/ prevname]
  ]  == end #LOOP
]  == end #DEF
```

Figure 3-14. Returning Information From a Release Handler (Page 3 of 3)

```
#SETMANY filespurged filesnotpurged exclude, 0 0 0
[#CASE [#ARGUMENT /VALUE filetemplate/ TEMPLATE &
    TOKEN /TOKEN !/]
  |1|
  |2|
    #SET exclude -1
    SINK [#ARGUMENT /VALUE filetemplate/ TEMPLATE]
]

[#LOOP |WHILE| 1 |DO|
  handletemplate
  [#CASE [#ARGUMENT /VALUE filetemplate/ COMMA TEMPLATE
      END]
    |1|
      #IF [#ARGUMENT /VALUE filetemplate/ TEMPLATE]
    |2|
    |3|
      #OUTPUT == blank line
      #OUTPUT Number of files purged    =[filespurged]
      #OUTPUT Number of files not purged=[filesnotpurged]
      #UNFRAME
      #RETURN
  ]
] == end of #LOOP
```

Creating a Keep Exception Handler

A keep exception handler processes exceptions but does not return to the calling process. If, for example, you want to provide a restrictive command shell with five commands, a keep handler allows you to process the five commands and any errors or break conditions without exiting the routine. The user could not, then, gain access to a standard TACL prompt.

The _CALL path is the entry point for the routine and, because control is to remain in the routine, it is not likely to be executed repeatedly. Therefore, the _CALL path contains the #FRAME and variable declarations that typically begin a routine.

Use the routine in Figure 3-15, `restricted_cmd_processor`, as a sample keep exception handler. If the BREAK key is pressed while the processing loop is running, TACL raises the _BREAK exception and reinvokes the routine; the #CASE function takes the _BREAK path and then reenters the loop.

If the user enters anything other than ADD or SUB, the #CASE statement in the loop raises _ERROR, and TACL reinvokes the routine; in this situation, the exception-processing #CASE takes the _ERROR path before resuming the loop.

Note The examples in this subsection include an EXIT case for testing purposes, which allows you to exit the routines. To prohibit exits, delete the EXIT case from the #FILTER statement and from the exception handler and main loop.

To invoke `restricted_cmd_processor`, load the file and enter:

```
restricted_cmd_processor
```

Figure 3-15. Sample Keep Exception Handler

```
?SECTION restricted_cmd_processor ROUTINE

[#CASE [#EXCEPTION]
  |_CALL |
    #FRAME
    #PUSH cmd
  |_BREAK|
    #OUTPUT BREAK key pressed.
  |_ERROR|
    #OUTPUT TACL error occurred.
  |EXIT|   == For demo only
    #RESET FRAMES RESULTS
    #RETURN
  |OTHERWISE|
    #OUTPUT Unknown exception occurred.
]  == End #CASE

== Filters predefined exceptions only
#FILTER _BREAK _ERROR EXIT

== After you enter this loop, control stays here unless the
== routine is processing an exception.
[#LOOP |DO|
  #SET cmd [#INPUT Enter cmd: ]  == Get value from terminal
  [#CASE [cmd]
    |ADD|
      #OUTPUT ADD
    |SUB|
      #OUTPUT SUB
    |EXIT|
      #RAISE EXIT              == For demo only
    |OTHERWISE|
      #OUTPUT Invalid command
      #RAISE _ERROR
  ]  == End CASE
|UNTIL| 0                      == Always false
]  == End LOOP
#UNFRAME
```

When you invoke restricted_cmd_processor, the output looks like this:

```
16> restricted_cmd_processor
Enter cmd: add
ADD
Enter cmd: clr
Invalid command
TACL error occurred.
Enter cmd: <BREAK>
BREAK key pressed.
Enter cmd: exit
17>
```

By using the definitions in Figure 3-16, a user can start and stop an application. (In the example, there is no code to start an application; the code performs a delay sequence to simulate application activity.)

The definitions in Figure 3-16 use the following global variables:

- Condition is an error flag; if 0, there is no error; if 1, there is an error, and recovery might be necessary.
- Recovery indicates the need to run a routine to clear local variables. If OFF, recovery is not necessary; if ON, recovery is necessary.

The shell supports the following commands:

- Coldstart —Pops old variables if necessary, pushes new variables, sets the condition and recovery variables, and then starts the application. If the user presses BREAK during this time, the exception handler sets the condition and recovery flags.

  When finished, coldstart resets the condition and recovery flags.
- Warmstart—Attempts to purge a file with an invalid file name; this attempt forces a TACL error to show exception handler operation.
- Shutdown —Performs a cleanup operation and sets condition to 0.
- Exit—Tests the condition variable and does not allow the user to exit until the error is resolved. If condition is 0, exit raises the user-defined EXIT exception and exits the shell.

Variables a, b, c, d, e, f, g, and h are created but are not used in this example; they are deleted during the cleanup phase before exiting.

If an error occurs during coldstart or warmstart, the user cannot exit the shell until a successful coldstart or shutdown occurs.

For this example, the keep handler returns if you enter an exit command. Usually, a keep handler would not provide an exit mechanism.

Figure 3-16. Sample Command Shell (Page 1 of 4)

```
?SECTION restrictive_command_shell ROUTINE
#PUSH #OUTFORMAT
#SET #OUTFORMAT PRETTY

== Define the code that handles exceptions
[#CASE [#EXCEPTION]
  |_CALL|
    #FRAME
    #PUSH err cmd prompt condition recovery
    #PUSH name step
    #SET condition 0
    display_initial_message
  |_BREAK|
    #OUTPUT /HOLD/ BREAK key hit during~_
    [#OUTPUT [#IF NOT [#EMPTYV name] |THEN|
        [name] at step [step]
      |ELSE|
         input
      ] == end of #IF
    ]  == end of #OUTPUT
    #SETMANY condition recovery , 1 REQUIRED
  |_ERROR|
    #OUTPUT Error occurred during [name] at step
    #OUTPUT [step]
    #ERRORTEXT /CAPTURE err/
    #OUTPUT The error is:
    #OUTPUTV  err
    #SETMANY condition recovery , 1 REQUIRED
  |EOF|
    #OUTPUT CTRL/Y will not break me
  |EXIT|
    #RESET FRAMES RESULTS
    #RETURN
]

== Body of the routine--enable four exception types:
#FILTER _BREAK _ERROR EOF EXIT

#OUTPUT
[#LOOP |DO|
  [#IF condition = 1 |THEN|
    #SET prompt An error condition exists--select &
       COLDSTART or SHUTDOWN:~_~_
  |ELSE|
     #SET prompt Select WARMSTART, COLDSTART, SHUTDOWN, &
 or EXIT:~_~_
  ]
```

Figure 3-16. Sample Command Shell (Page 2 of 4)

```
    #INPUTV /UNTIL TACL/ cmd prompt
    [#IF ([#INPUTEOF]) |THEN|
        #RAISE EOF
    ]
    [#CASE [cmd]
      |WARMSTART|
        warmstart
      |COLDSTART|
        coldstart
      |SHUTDOWN|
        shutdown
      |EXIT|
        EXIT
      |OTHERWISE |
        #OUTPUT Invalid command
    ]
  |UNTIL| 0 = 1
  ] == end #LOOP
  #UNFRAME

  ?SECTION display_initial_message TEXT
  #OUTPUT This interface is used to:
  #OUTPUT COLDSTART, WARMSTART, or SHUTDOWN the application.
  #OUTPUT
  #OUTPUT EXIT is disabled if an error exists.
  #OUTPUT The BREAK key and CTRL/Y do not cause an EXIT.

  ?SECTION warmstart MACRO  == WARMSTART the application
  [#IF condition = 1 |THEN|
    #OUTPUT An error condition exists.
    #OUTPUT Must COLDSTART or SHUTDOWN.
    #OUTPUT
  |ELSE|
    [#CASE [recovery]
      |REQUIRED|
        cleanup
      | OTHERWISE |
    ]
    #SET name [#VARIABLEINFO /VARIABLE/ %0%]
    #SET step 1
    #PUSH a b
    #DELAY 200
    #OUTPUT WARMSTARTing application $X
```

Figure 3-16. Sample Command Shell (Page 3 of 4)

```
    #SET step 2
    #PUSH c d
    #DELAY 200
    #PURGE filewithlongname
    #OUTPUT
  ]  == end #IF

  ?SECTION coldstart MACRO  == COLDSTART the application
  [#CASE [recovery]
    |REQUIRED  |
      cleanup
    |OTHERWISE |
  ]

  #SET name [#VARIABLEINFO /VARIABLE/ %0%]
  #SET step 1
  #PUSH e f

  == If an error occurs during this step, force recovery
  #OUTPUT COLDSTARTing application $X
  #OUTPUT (Press BREAK now to force a recovery)
  #DELAY 300
  #SETMANY condition recovery , 0 OFF
  #OUTPUT COLDSTART Successful
  #OUTPUT

  ?SECTION shutdown MACRO  == SHUTDOWN the application
  #OUTPUT SHUTDOWN of application $X
  [#CASE [recovery]
    |REQUIRED|
      cleanup
    |OTHERWISE|
  ]

  #SET name [#VARIABLEINFO /VARIABLE/ %0%]
  #SET step 1
  #PUSH g h
  #DELAY 300
  #SET condition 0
  #OUTPUT SHUTDOWN Successful
  #OUTPUT
```

Figure 3-16. Sample Command Shell (Page 4 of 4)

```
?SECTION exit MACRO  == EXIT the exception handler
[#IF condition = 1 |THEN|
  #OUTPUT Can not EXIT without resolving the error.
  #OUTPUT Must WARMSTART, COLDSTART, or SHUTDOWN.
  #OUTPUT
|ELSE|
  #OUTPUT Exiting restrictive command shell.
  #RAISE EXIT
]

?SECTION cleanup MACRO == delete variables from prev. errors
#SET #BREAKMODE DISABLE
#OUTPUT Performing cleanup procedure
#OUTPUT The Break Key is disabled until cleanup is complete.
[#CASE [name][step]
  | WARMSTART1 |
    #POP a b
  | WARMSTART2 |
    #POP a b c d
  | COLDSTART1 |
    #POP e f
  | SHUTDOWN1  |
    #POP g h
]

#OUTPUT
#DELAY 200
#SETMANY condition recovery , 0 OFF
#SET #BREAKMODE ENABLE
#OUTPUT Cleanup procedure complete.
```

Combining Keep and Release Handlers

The routines in Figure 3-17 show one way to combine a keep handler (`restricted_caller`) and a release handler (`protected_code`). `Restricted_caller` starts first; therefore, if an exception occurs, control returns to the processing loop after the exception is processed.

When the user enters a valid command, `restricted_caller` calls `protected_code` to execute the command; that routine, in turn, calls either `do_add` or `do_sub`. If the user presses the BREAK key or an unknown exception is raised during execution of either of the latter routines, TACL pops the routine and reinvokes `protected_code`. The #CASE function takes the OTHERWISE path, which performs an orderly deallocation of resources and then raises the same exception.

TACL then pops that routine (the #FILTER function has not yet been executed in the reinvocation) and returns to `restricted_caller`. The #CASE function in that routine takes the appropriate path to deal with the exception and restarts the processing loop.

For this example, the keep handler terminates if you enter an EXIT command. Usually, a keep handler does not provide an exit mechanism.

Figure 3-17. Using Nested Keep and Release Handlers (Page 1 of 2)

```
?SECTION restricted_caller ROUTINE
[#CASE [#EXCEPTION]
  |_CALL|
    #FRAME
    #PUSH cmd exceptionlist
    #SET exceptionlist _BREAK _ERROR EXIT
  |_BREAK|
    #OUTPUT BREAK key pressed.
  |_ERROR|
    #OUTPUT TACL error occurred.
  | EXIT |  == for demonstration purposes
    #RESET FRAMES RESULTS
    #RETURN
  |OTHERWISE|
    #OUTPUT Unknown exception occurred.
]  == End CASE

== Filter for predefined exceptions only
#FILTER [exceptionlist]
[#LOOP |DO|
  #SET cmd [#INPUT Enter cmd: ]  == Get value from terminal
  [#CASE [cmd]
    |ADD|
      protected_code do_add
    |SUB|
      protected_code do_sub
    |EXIT|  == For demonstration purposes
      #RAISE EXIT
    |OTHERWISE|
      #OUTPUT Invalid command
      #RAISE _ERROR
  ]  == End CASE
|UNTIL| 0 = 1 ==  (do forever)
]  == End LOOP
#UNFRAME
```

Figure 3-17. Using Nested Keep and Release Handlers (Page 2 of 2)

```
?SECTION protected_code ROUTINE
[#CASE [#EXCEPTION]
  |_CALL |
    == No action required when first called
  |OTHERWISE|
    #RESET FRAMES RESULTS == Deallocate resources
    #RAISE [#EXCEPTION]   == Return to caller with
                          == exception raised, so that
                          == the caller executes its
                          == exception handler
]  == End CASE

#FILTER [exceptionlist]
[#REST]  == Invoke rest of arguments (call macro)

?SECTION do_add ROUTINE
#OUTPUT Adding

?SECTION do_sub ROUTINE
#OUTPUT Subtracting
```

4 Accessing Files

The #REQUESTER built-in function allows you to open a file, process, or device so that you can send messages or records to it or read messages or records from it.

This section describes how to use #REQUESTER to access files. For information about the use of #REQUESTER with processes, see "Using $RECEIVE" in Section 5, "Initiating and Communicating With Processes."

#REQUESTER Operation

To open a file, call the #REQUESTER function and include the file name and a set of variables that are used to transmit data. If you plan to set up more than one #REQUESTER operation, you can identify variables by including file identification information in each variable name. To list a variable and its association with the #REQUESTER operation, use the VARINFO command.

The call to #REQUESTER does not perform input or output; it opens the specified file and initializes the associated variables. If a file system error occurs during this step, #REQUESTER returns the error.

The #REQUESTER function opens a file for waited or nowaited I/O. After you invoke the #REQUESTER function, TACL continues to execute code. For waited I/O operations, TACL stops at the next I/O request and ensures that each read or write is complete before processing the next request. For nowaited operations, call #WAIT to determine whether your request has been completed. If you plan to read or write records larger than 239 bytes, you must use waited I/O.

To initiate a read or write operation, you append data to the appropriate variable, as described in the following subsections. When your TACL process first detects data in the variable, TACL initiates the operation and transfers a record of data.

To read and write from the same file, call #REQUESTER twice to establish two communication paths to the file. Use a separate set of variables for each communication path.

When you use #REQUESTER, your TACL process does not create a separate process, but manages the I/O from within your TACL process. The #REQUESTER function uses sequential I/O to access files, devices, and processes.

Note The way in which you order the variables in the #REQUESTER call is very important; the file name must be first, followed by the error variable and the read or write variable. For a read operation, the prompt variable must be specified last.

To close a file, call #REQUESTER with the CLOSE option.

Table 4-1 lists functions related to #REQUESTER operation.

Table 4-1. Functions Used With #REQUESTER

Function	Description
#APPEND, #APPENDV	Adds lines to a variable.
#EXTRACT, #EXTRACTV	Retrieves lines from a variable.

Requesting Waited Reads

To open a file for waited read operations, issue a #REQUESTER call and include the WAIT option; for example, the following statement opens FILE1 and initializes `error_var`, `read_var`, and `prompt_var`:

```
#SET rslt [#REQUESTER /WAIT/ READ file1 error_var read_var
  prompt_var]
```

You can also use the WAIT option to specify the size of the text buffer. To specify shared, protected, or exclusive access to the file, use the EXCLUSION option; the default for a read operation is shared. For example:

```
#REQUESTER /EXCLUSION PROTECTED/ READ file2 error_var &
  read_var prompt_var
```

Note It is very important to check the results of the open operation; otherwise, you will not know if the open request received an error.

To initiate a read operation, append data to the prompt variable:

```
#APPEND prompt_var *start read*
```

When reading a disk file, TACL discards the data in `prompt_var`; you can specify any non-null data. TACL reads a record from FILE1 and places it into `read_var`. You can use #EXTRACT(V) to retrieve data from `read_var`; as you #EXTRACT records, TACL deletes them from `read_var`.

Each time you append a line to `prompt_var`, the TACL process reads a record from the disk file FILE1 and appends it to `read_var`. TACL then performs a READ operation. TACL continues executing code until it encounters an #APPEND(V) or #EXTRACT(V) call that refers to one of the #REQUESTER variables. TACL then waits until the current read operation is complete before initiating the next read operation.

When you are finished reading from the file, issue a CLOSE request and supply one of the variable levels associated with the file; for example:

```
#REQUESTER CLOSE read_var
```

This operation closes FILE1 (associated with `read_var`) and terminates the #REQUESTER function.

Use the routine in Figure 4-1, `waited_read`, to perform waited reads from the file specified in the first argument in the invocation and display the records on the terminal. To invoke this routine, load the file and type:

```
waited_read filename
```

The routine stops when it detects an error or end-of-file.

Figure 4-1. Performing a Waited Read

```
?SECTION waited_read ROUTINE
#FRAME
#PUSH open_error read_error read_data read_prompt line
#PUSH read_file rslt

#SET rslt [#ARGUMENT /VALUE read_file/ FILENAME OTHERWISE]
[#CASE [rslt]
  |1|
    == Open the file:
    #SET open_error [#REQUESTER /WAIT/ READ [read_file]
                 read_error read_data read_prompt]

    [#IF [open_error] |THEN|
      #OUTPUT *** Error opening [read_file]:  [open_error]
      #RETURN
    ]
    #SET read_error 0
    [#LOOP |WHILE| NOT [read_error] |DO|
      #APPEND read_prompt *start*
      #EXTRACTV read_data line
      #OUTPUTV line
    ]

  |OTHERWISE|
    #OUTPUT *** Error:  Invalid file ***
    #RETURN
] == end #CASE

[#IF read_error = 1 |THEN|
  #OUTPUT *** End of file ***
|ELSE|
  #OUTPUT *** Error reading [read_file]:  [read_error]
]

SINK [#REQUESTER/WAIT/CLOSE read_data]
#UNFRAME
```

Requesting Nowaited Reads

To open a file for nowaited read operations, issue a #REQUESTER call and omit the WAIT option; for example, the following statement opens FILE1 and initializes `error_var`, `read_var`, and `prompt_var`:

```
#SET rslt [#REQUESTER READ file1 error_var read_var
  prompt_var]
```

Note It is very important to check the results of the open operation; otherwise, you will not know if the open request received an error.

To specify shared, protected, or exclusive access to the file, use the EXCLUSION option; the default for a read operation is shared. For example:

```
#REQUESTER /EXCLUSION PROTECTED/ READ file2 error_var &
  read_var prompt_var
```

To initiate a read operation, append data to the prompt variable:

```
#APPEND prompt_var *start read*
```

To read a disk file, TACL discards the data in `prompt_var`, reads a record from FILE1, and places it into `read_var`. You can use #EXTRACT(V) to retrieve data from `read_var`; as you #EXTRACT records, TACL deletes them from `read_var`.

Each time you append a line to `prompt_var`, the TACL process reads a record from disk file FILE1 and appends it to `read_var`. TACL then performs a READ operation. TACL continues executing code; when you are ready to wait for completion of the read operation, use the #WAIT built-in function to wait until `read_var` contains data. To avoid writing over data that has not yet been transmitted, use #WAIT to make sure the previous operation has finished.

When you are finished reading from the file, issue a CLOSE request and supply one of the variables associated with the file; for example:

```
#REQUESTER CLOSE error_var
```

This operation closes FILE1 (associated with `error_var`) and terminates the #REQUESTER function.

Use the routine in Figure 4-2, `nowaited_read`, to perform nowaited reads from the file specified in the first argument in the invocation and display the records on the terminal. To invoke this routine, load the file and type:

```
nowaited_read filename
```

This routine uses #VARIABLEINFO/VARIABLE/ to return a variable name to the #CASE statement. #VARIABLEINFO/VARIABLE/ returns the name of a variable without the level number so that it will match one of the labels. The #WAIT built-in function, when used alone, returns the variable name with the level number.

Figure 4-2. Performing a Nowaited Read

```
?SECTION nowaited_read ROUTINE
#FRAME
#PUSH open_error read_error read_data read_prompt line
#PUSH read_file rslt

#SET rslt [#ARGUMENT /VALUE read_file/ FILENAME OTHERWISE]
[#CASE [rslt]
  |1|
    == Open the file:
    #SET open_error [#REQUESTER /WAIT/ READ [read_file]
                     read_error read_data read_prompt]

    [#IF [open_error] |THEN|
      #OUTPUT *** Error opening [read_file]:  [open_error]
      #RETURN
    ]
    #SET read_error 0
    [#LOOP |WHILE| NOT [read_error] |DO|
      #APPEND read_prompt *start*
      [#CASE [#VARIABLEINFO/VARIABLE/
          [#WAIT read_data read_error]] |THEN|
        |read_data |
          #EXTRACTV read_data line
          #OUTPUTV line
        |read_error|
           == exit loop
      ]  == end case
    ]  == end loop
  |OTHERWISE|
    #OUTPUT *** Error:  Invalid file ***
    #RETURN
] == end #CASE

[#IF read_error = 1 |THEN|
  #OUTPUT *** End of file ***
|ELSE|
  #OUTPUT *** Error reading [read_file]:  [read_error]
]

SINK [#REQUESTER/WAIT/CLOSE read_data]
#UNFRAME
```

Requesting Waited Writes

To open a file for waited write operations, issue a #REQUESTER call and include the WAIT option. The following statement opens FILE1 and initializes `error_var` and `write_var`. If FILE1 does not exist, TACL creates an Edit file:

```
#SET rslt [#REQUESTER /WAIT/ WRITE file1 error_var write_var]
```

Note It is very important to check the results of the open operation. Otherwise, you will not know if the open request received an error.

You can also use the WAIT option to specify the size of the text buffer. To specify shared, protected, or exclusive access to the file, use the EXCLUSION option; the default for a write operation is shared. For example:

```
#REQUESTER /EXCLUSION PROTECTED/ WRITE file2 error_var &
  write_var
```

To initiate the write operation, append data to the write variable:

```
#APPEND write_var This is a test
```

When TACL detects data in `write_var`, it writes the data to FILE1.

You can add a record to a structured file but cannot replace a record. If you attempt to write a record that already exists, TACL returns an error.

Each time you append a line to `write_var`, the TACL process writes the record to FILE1. TACL continues executing code until it encounters an #APPEND(V) call that refers to one of the #REQUESTER variables, signifying that you have more data to write. TACL then waits until the current operation is complete before initiating the next write operation.

When you are finished writing to the file, issue a CLOSE request and supply one of the variable levels associated with the file; for example:

```
#REQUESTER CLOSE error_var
```

This operation closes FILE1 (associated with `error_var`) and terminates the #REQUESTER function.

Use the routine in Figure 4-3, `waited_write`, to perform waited writes to the file specified as the first argument in the invocation. If the file already contains data, this routine appends the new data to the end of the file. To invoke this routine , load the file and type:

```
waited_write filename
```

The #INPUT call in Figure 4-3 reads a line from the TACL IN file.

Figure 4-3. Reading From a Terminal and Performing a Waited Write

```
?SECTION waited_write ROUTINE
#FRAME
#PUSH open_error write_error write_data write_prompt line
#PUSH write_file rslt
#SET #INPUTEOF 0

#SET rslt [#ARGUMENT /VALUE write_file/ FILENAME OTHERWISE]
[#CASE [rslt]
  |1|
    == Open the file:
    #SET open_error [#REQUESTER/WAIT/WRITE [write_file]
                     write_error write_data]
    [#IF [open_error] |THEN|
      #OUTPUT *** Error opening [write_file]:  [open_error]
      #RETURN
    ]
    [#LOOP |DO|
      #APPEND write_data [#INPUT a_:]
      == Add error checking here for output file if necessary
    |UNTIL| [#INPUTEOF]
    ]
  |OTHERWISE|
    #OUTPUT *** Error:  Invalid file ***
    #RETURN
] == end #CASE

SINK [#REQUESTER/WAIT/CLOSE write_data]
#UNFRAME
```

Requesting Nowaited Writes

To open a file for nowaited write operations, issue a #REQUESTER call and omit the WAIT option. The following statement opens FILE1 and initializes `error_var` and `write_var`. If FILE1 does not exist, TACL creates an Edit file:

```
#SET rslt [#REQUESTER WRITE file1 error_var write_var]
```

Note It is very important to check the results of the open operation. Otherwise, you will not know if the open request received an error.

To specify shared, protected, or exclusive access to the file, use the EXCLUSION option; the default for a write operation is shared. For example:

```
#REQUESTER /EXCLUSION PROTECTED/ WRITE file2 error_var &
  write_var
```

To initiate the write operation, append data to the write variable:

```
#APPEND write_var This is a test
```

When TACL detects data in `write_var`, it writes the record to FILE1.

You can add a record to a structured file but you cannot replace a record. If you attempt to write a record that already exists, TACL returns an error.

Each time you append a line to `write_var`, the TACL process writes a record to FILE1. TACL continues executing code. When you are ready to wait for completion of the read operation, use the #WAIT built-in function to wait until the `write_var` contains data. To avoid writing over data that has not yet been transmitted, use #WAIT to make sure the previous operation has finished.

When you are finished writing to the file, call the #WAIT function to make sure that the last write has finished. Next, issue a CLOSE request and supply one of the variable levels associated with the file; for example:

```
#REQUESTER CLOSE write_var
```

This operation closes FILE1 (associated with `write_var`) and terminates the #REQUESTER function.

Use the routine in Figure 4-4, nowaited_write, to perform nowaited writes to the file specified in the first argument in the invocation. If the file already contains data, this routine appends the new data to the end of the file. To invoke this routine, load the file and type:

```
nowaited_write filename
```

Figure 4-4. Reading From a Terminal and Performing a Nowaited Write

```
?SECTION nowaited_write ROUTINE
#FRAME
#PUSH open_error write_error write_data write_prompt line
#PUSH write_file rslt
#SET #INPUTEOF 0

#SET rslt [#ARGUMENT /VALUE write_file/ FILENAME OTHERWISE]
[#CASE [rslt]
  |1|
    == Open the file:
    #SET open_error [#REQUESTER WRITE [write_file]
                       write_error write_data]
    [#IF [open_error] |THEN|
      #OUTPUT *** Error opening [write_file]:  [open_error]
      #RETURN
    ]
    [#LOOP |DO|
      [#CASE [#VARIABLEINFO/VARIABLE/
          [#WAIT write_data write_error]]
        |write_data |
          #APPEND write_data [#INPUT b_:]
        |write_error|
          #OUTPUT Error [write_error] on write
      ]
    |UNTIL| [#INPUTEOF]
    ]
  |OTHERWISE|
    #OUTPUT *** Error:  Invalid file ***
    #RETURN
] == end #CASE

SINK [#REQUESTER/WAIT/CLOSE write_data]
#UNFRAME
```

Copying Records Between Files

Use the routine in Figure 4-5, `copy`, to read records from one file and write them to another file. The source file is specified as the first argument; the destination file is specified as the second argument; both files must exist. In this example:

- `read_err` is the error variable for the read operation
- `read_var` contains the data obtained from the read operation
- `prompt_var` is the prompt variable to start the read operation
- `write_err` is the error variable for the write operation
- `write_var` contains the record to be written

To invoke this routine, load the file that contains `copy` and then type:

```
copy file1 file2
```

`Copy` appends data to the end of FILE2. `Copy` could be changed to modify records before copying them; for example, you could search for a string and, if it is present, modify the string before writing the record to the destination file.

Figure 4-5. Copying Records From One File to Another File (Page 1 of 2)

```
?SECTION copy ROUTINE
#FRAME
#PUSH rslt rslt2 source dest
#PUSH write_err write_var read_err read_var prompt_var
#PUSH Ready open_err

== Check for existence of the first argument.  If empty,
== display a message and exit.

#SET rslt [#ARGUMENT /VALUE source/ FILENAME OTHERWISE]
[#CASE [rslt]
  |1|
    == Check for the second file.
    #SET rslt2 [#ARGUMENT /VALUE dest/ FILENAME OTHERWISE]
    [#CASE [rslt2]
      |1|
        == Open the source and destination files and
        == associate variables with them.
        #SET open_err [#REQUESTER READ [source] read_err
            read_var prompt_var]
        [#IF open_err = 0 |THEN|
          #OUTPUT [source] opened successfully
        |ELSE|
          #OUTPUT [source] not open; error [read_err]
          #RETURN
        ]
        #SET open_err [#REQUESTER WRITE [dest] write_err
            write_var]
```

Figure 4-5. Copying Records From One File to Another File (Page 2 of 2)

```
            [#IF open_err = 0 |THEN|
              #OUTPUT [dest] opened successfully
            |ELSE|
              #OUTPUT [dest] not open; error [write_err]
              SINK [#REQUESTER CLOSE read_var]   == close source
              #RETURN
            ]

            == Initiate read and write operations.
            #SET read_err 0   == initialize read_err
            [#LOOP |DO|
              == Start the read.
              #APPEND prompt_var READIT
              == Wait for read_var or read_err to change.
              #SET ready [#WAIT read_var read_err]
              == If read_var changed, the read was successful.
              [#IF [#MATCH read_var.* [Ready]] |THEN|
                == Wait for the last write to complete
                SINK [#WAIT write_var]
                == Move the record into write_var to initiate the
                == write operation.
                #EXTRACTV read_var write_var
                ]
            |UNTIL| ([read_err])
            ]  == end of #loop

            == Wait for the last write operation to finish.
            SINK [#WAIT write_var]

            == Close both files and terminate the #REQUESTER
            == functions.
            SINK [#REQUESTER CLOSE read_var]
            SINK [#REQUESTER CLOSE write_var]
          |2|
            #OUTPUT *** Error:  Invalid destination filename ***
          ]
      |2|
        #OUTPUT *** Error:  Invalid source filename ***
      ]
    #UNFRAME
```

The #EXTRACTV call that performs the write operation clears the contents of `read_var` and `write_var`. TACL moves a record out of `read_var` and into `write_var`. After TACL writes the data, TACL deletes the record from `write_var`.

Comparing Files

Use the routine in Figure 4-6, `fcomp`, to perform a line-for-line comparison of two files. A mismatch does not resynchronize the two files.

`Fcomp` reads a record from each of the two files and then calls #COMPAREV to compare the records. The maximum line length for an edit file is 239 bytes; `fcomp` uses this value as a maximum line length for the input records.

`Fcomp` supports two options: you can limit the comparison to a range of columns within the files, and you can write the results to a file.

`Fcomp` calls the `getargs` macro (from Section 3, "Developing TACL Routines") and the `defaultvars` macro (from Section 2, "Developing TACL Programs"):

- `Getargs` parses the arguments of a calling routine; it accepts sets of three arguments:
  - REQUIRED or OPTIONAL specify whether an argument must be present or can be omitted from the list of arguments.
  - `Type` specifies an #ARGUMENT alternative, such as FILENAME or KEYWORD.
  - `Variable` is pushed and set by the TEXT option of #ARGUMENT if the argument is supplied; otherwise, the variable remains empty.
- `defaultvars` accepts a space-separated list of space-separated pairs (variable levels and values) and sets each empty variable level to its corresponding value. For each pair of arguments:
  - *`Variable-level`* specifies the name of a variable level.
  - *`Value`* specifies the corresponding value, or can be empty. Value cannot contain any spaces.

To call `fcomp` , load the associated file and enter:

```
fcomp file1, file2 [, [f1] [, [f2] [, [result]]]]
```

where

`file1`

is the name of one of the comparison files.

`file2`

is the name of the other comparison file.

`f1`

is a starting field range (optional).

`f2`

is an ending field range (optional).

`result`

is a file that will contain the results of the comparison (optional).

Figure 4-6. Comparing Two Files (Page 1 of 3)

```
?SECTION fcomp ROUTINE
#FRAME

== Define a character array; the maximum line length is 239
== characters.
#DEF char_array STRUCT BEGIN CHAR column(1:239); END;

== Use the character array to define two STRUCTs that contain
== character arrays. Fcomp uses these STRUCTS to compare
== lines of data.
#DEF line1 STRUCT LIKE char_array;
#DEF line2 STRUCT LIKE char_array;
#PUSH linecount file1_err file1_var file1_prompt
#PUSH file2_err file2_var file2_prompt

#SET linecount 0

== Call the getargs macro to parse FCOMP arguments
[getargs  required FILENAME file1
          required FILENAME file2
          optional NUMBER f1
          optional NUMBER f2
          optional FILENAME/SYNTAX/ results
]

== Call defaultvars to set defaults if parts of the column
== range were not specified.
[defaultvars    f1 1 f2 239]

== Check to make sure the column range is between 1 and 239.
== If not, display a message and exit.
[#IF (f1<1) OR (f1>239) OR (f2<1) OR (f2>239) OR (f1>f2)
|THEN|
  #OUTPUT Illegal field range value
  #UNFRAME
  #RETURN
]
```

Figure 4-6. Comparing Two Files (Page 2 of 3)

```
== If a result file was specified, use it for output.
== Otherwise, use the default OUT file.
[#IF NOT [#EMPTYV results] |THEN|
  #PUSH #OUT
  #SET #OUT [results]
]

== Display the user's request.
#OUTPUT
#OUTPUT *-------------------------------------------------*
#OUTPUT Files compared: [file1],[file2] -- Field:<[f1]:[f2]>
#OUTPUT *-------------------------------------------------*
== Open the files.
#SET file1_err [#REQUESTER /WAIT/ READ [file1] file1_err &
  file1_var file1_prompt]
[#IF file1_err = 0 |THEN|
  #OUTPUT [file1] opened successfully
|ELSE|
  #OUTPUT [file1] not open; file error [file1_err]
  #RETURN
]
#SET file2_err [#REQUESTER/WAIT/ READ [file2] file2_err &
  file2_var file2_prompt]
[#IF file2_err = 0 |THEN|
  #OUTPUT [file2] opened successfully
|ELSE|
  #OUTPUT [file2] not open; file error [file2_err]
  SINK [#REQUESTER CLOSE file1_err]  == close FILE1
  #RETURN
]

== While there are no errors, read and compare lines.
[#LOOP |WHILE| file1_err = 0 AND file2_err = 0 |DO|

  == Increment the line counter by 1.
  #SET linecount [#COMPUTE linecount + 1]

  == Read a line from the first file
  #SET file1_prompt  *start*
  #EXTRACTV file1_var line1
```

Figure 4-6. Comparing Two Files (Page 3 of 3)

```
    == If the first read was successful, read a line from the
    == second file.
    [#IF file1_err <= 1 |THEN|
      #SET file2_prompt *start*
      #EXTRACTV file2_var line2
      == If the second read was successful, perform the
      == comparison.
      [#IF NOT file2_err |THEN|
        == If the two lines do not match, display them.
        [#IF NOT [#COMPAREV line1:column([f1]:[f2])
          line2:column([f1]:[f2])] |THEN|
          #OUTPUT File [file1] line [linecount]: [line1]
          #OUTPUT *--------------------------------------*
          #OUTPUT File [file2] line [linecount]: [line2]
          #OUTPUT *--------------------------------------*
        ] == end of #IF ... #COMPAREV
      ] == end of #IF NOT file2_err
    ] == end of #IF file1_err
  ] == end of #LOOP

  == Close the files
  SINK [#REQUESTER CLOSE file1_err]
  SINK [#REQUESTER CLOSE file2_err]

  == Handle EOF conditions, file size mismatches, and errors.
  [#CASE [file1_err]^[file2_err]
    |1^1| #OUTPUT [linecount] lines compared    == Normal exit;
    |0^1| #OUTPUT [file1] is has more lines than [file2]
    |1^0| #OUTPUT [file2] is has more lines than [file1]
    |OTHERWISE|
      [#IF file1_err > 1 |THEN|
        #OUTPUT Error [file1_err] on [file1]
      ]
      [#IF file2_err > 1 |THEN|
        #OUTPUT Error [file2_err] on [file2]
      ]
  ]
  #UNFRAME
```

Listing a File

Use the macro in Figure 4-7, `tacllist`, to format, paginate, and print TACL program files. This macro calls #REQUESTER to read the TACL file, copies the TACL file to a disk file designated as the OUT file in TFORM format, and then calls TFORM to format and print the file. `Tacllist` displays the following:

- A listing banner with the TACL file date and the current date.
- The contents of the file, including line numbers and page numbers.

Note Figure 4-7 shows the use of a macro for a more complex set of operations. Because a macro cannot use the #ARGUMENT built-in function, the macro must provide more argument checking capabilities. In addition, the macro cannot use the #RETURN built-in function, so it uses a series of #IF calls. The bulk of the code resides in the innermost #IF statement, making the program more difficult to read.

You can use the TFORM \NEW command within the TACL file to cause TFORM to advance to the top of a new page. When you insert a directive of the form ==\NEW in the file to be printed, `tacllist` replaces it with \NEW. The macro sets the page width to 132 characters.

To preview your output before printing, specify your terminal name as the output file. Following a preview, reinvoke `tacllist` to get a printed listing. To run this macro, load the file and enter:

```
tacllist infilename [outfilename]
```

If `outfilename` already exists, TACL purges the file. If you do not specify a result file, `tacllist` writes to a file called TACLTFRM.

Figure 4-7. Listing a File (Page 1 of 5)

```
?SECTION tacllist MACRO

#FRAME
#PUSH err_inp rec_inp prompt default_outfile
#PUSH line_num line page_num page_out bin_date file_date
#PUSH list_date year month day file_hour file_min
#PUSH list_hour list_min short_stars long_stars
#PUSH lines_out max_lines match_string printer
#PUSH #OUTFORMAT #WIDTH

#SET printer $S.#LP5          == Put local printer name here
#SET max_lines  55            == Set to maximum lines/page
#SET default_outfile TACLTFRM  == Default output file name
#SET #WIDTH 132               == Listing wrap column
#SET #OUTFORMAT PLAIN
```

Figure 4-7. Listing a File (Page 2 of 5)

```
[#DEF file STRUCT
  BEGIN
  CHAR input (0:33);
  CHAR output(0:33);
  END;
] == end DEF

==  Output a banner
[#DEF output_banner TEXT |BODY|
  #SET short_stars *********
  #SET long_stars **********************************
  #OUTPUT [long_stars] tacllist [long_stars]
  #OUTPUT/HOLD                                          / &
    *** Program Name:  [file:input(0:33)]
  #OUTPUT/HOLD,COLUMN 49                                / &
    Listing Date:  [list_date]
  #OUTPUT/HOLD,COLUMN 71,WIDTH 2,JUSTIFY RIGHT,FILL ZERO/ &
    [list_hour]
  #OUTPUT/HOLD,COLUMN 73,WIDTH 1                        / &
    :
  #OUTPUT/HOLD,COLUMN 74,WIDTH 2,JUSTIFY RIGHT,FILL ZERO/ &
    [list_min]
  #OUTPUT/     COLUMN 77                                / &
    ***
  #OUTPUT/HOLD                                          / &
    *** Program Date:  [file_date]
  #OUTPUT/HOLD,COLUMN 27,WIDTH 2,JUSTIFY RIGHT,FILL ZERO/ &
    [file_hour]
  #OUTPUT/HOLD,COLUMN 29,WIDTH 1                        / &
    :
  #OUTPUT/HOLD,COLUMN 30,WIDTH 2,JUSTIFY RIGHT,FILL ZERO/ &
    [file_min]
  #OUTPUT/HOLD,COLUMN 49                                / &
    Page Number:
  #OUTPUT/HOLD,COLUMN 73,WIDTH 3,JUSTIFY RIGHT,FILL ZERO/ &
    [page_num]
  #OUTPUT/     COLUMN 77                                / &
    ***
  #OUTPUT [long_stars][short_stars][long_stars]
  #OUTPUT
  #SET page_num [#COMPUTE page_num + 1]
] == end DEF
```

Figure 4-7. Listing a File (Page 3 of 5)

```
== Get dates macro
[#DEF get_dates TEXT |BODY|
  #SET bin_date [#FILEINFO/MODIFICATION/ [file:input(0:33)]]
  #SETMANY year month day file_hour file_min, &
    [#CONTIME [bin_date]]
  #SET file_date &
    [#COMPUTE ((year - 1900) * 100000) + (month * 100) + day]
  #SETMANY year month day list_hour list_min, &
    [#CONTIME [#TIMESTAMP]]
  #SET list_date &
    [#COMPUTE ((year - 1900) * 100000) + (month * 100) + day]
]

== Main Part of the Code
============================================================
== Look for the input file name.  If empty, drop out of the
== loop and display an error.
[#IF NOT [#EMPTY %1%] |THEN|
  #SET file:input(0:33) [#SHIFTSTRING/UP/%1%]

  == Look for the out file name. If empty, use the default
  == OUT file.
  [#IF NOT [#EMPTY %2%] |THEN|
    #SET file:output(0:33) [#SHIFTSTRING/UP/%2%]
  |ELSE|
    #SET file:output(0:33) [default_outfile]
  ]
  == Open the input file for read access.
  #SET err_inp [#REQUESTER/WAIT/READ [file:input(0:33)] &
    err_inp rec_inp prompt]
  [#IF err_inp <> 0 |THEN|  == open error; drop out of loop.
    #OUTPUT [file:input(0:33)] not open; error: &
      [err_inp]
  |ELSE|
    == If the output file already exists, purge it.
    [#IF [#FILEINFO/EXISTENCE/ [file:output(0:33)]] |THEN|
      SINK [#PURGE [file:output(0:33)] ]
    ]
    == Save (push) the current #OUT setting, then set it to
    == the new value.
    #PUSH #OUT
    #SET #OUT [file:output(0:33)]
```

Figure 4-7. Listing a File (Page 4 of 5)

```
== The following variables keep track of the line number,
== page number, and number of lines already written to
== a page (for calculation of automatic page break).
#SET line_num  1
#SET page_num  1
#SET lines_out 0
#SET match_string \NEW
#OUTPUT \STYLE FORM WIDTH 132 CHARS
get_dates
output_banner

== Start the read operation; place the record into line.
#APPEND prompt *start*
#EXTRACTV rec_inp line

[#LOOP |WHILE| NOT err_inp |DO|
  == If the line equals \NEW, perform a page break.
  [#IF [#MATCH *[match_string]* [line]] |THEN|
    #OUTPUT \NEW
    output_banner
    #SET lines_out 0
  == Otherwise, output the line number and the line.
  |ELSE|
    #OUTPUT/HOLD,WIDTH 3,COLUMN 1,JUSTIFY RIGHT,FILL &
      ZERO/ [line_num]
    #OUTPUT/HOLD,WIDTH 1/
    #OUTPUTV line
    #SET lines_out [#COMPUTE lines_out + 1]
  ]

== If the page is full, issue a page break.
  [#IF (lines_out = max_lines) |THEN|
    #OUTPUT \NEW
    output_banner
    #SET lines_out 0
  ]
  == Move the next record into line.
  #APPEND prompt *start*
  #EXTRACTV rec_inp line

  == Increment the line counter.
  #SET line_num [#COMPUTE line_num + 1]
] == end LOOP
```

Figure 4-7. Listing a File (Page 5 of 5)

```
      [#IF err_inp <> 1 |THEN|      == Error 1 is EOF (OK)
        #OUTPUT Error [err_inp]: Opening/reading file &
          [file:input(0:33)]
      ]
      #SET err_inp [#REQUESTER/WAIT/CLOSE rec_inp]
      #POP #OUT
      [#IF [#FILEINFO/CODE/ [file:output(0:33)]] = 101 |THEN|
        TFORM/IN [file:output(0:33)], OUT [printer], NOWAIT/
      ]
    ] == end of #IF [err_inp] <> 0
  |ELSE| == Error in input parameters
    #OUTPUT
    #OUTPUT ERROR:  Missing input parameter.  Format is:
    #OUTPUT
    #OUTPUT/COLUMN 3/TACLLIST inputfilename~[outputfilename~]
    #OUTPUT
  ] == end of #IF NOT [#EMPTY %1%]
  #UNFRAME
```

Figure 4-8 contains a sample listing produced by the TACLLIST macro.

Figure 4-8. TACLLIST Output

```
\STYLE FORM WIDTH 132 CHARS
************************** tacllist ************************
*** Program Name: S6A          Listing Date: 920415 12:21 ***
*** Program Date: 920415 12:20 Page Number:           001 ***
************************************************************

001 ?SECTION defaultvars MACRO
002
003 == Any more pairs?
004 [#IF NOT [#EMPTY %1%] |THEN|
005
006   == Is this variable level empty?
007   [#IF [#EMPTYV %1%] |THEN|
008
009     == The variable level is empty; install default value
010     #SET %1% %2%
011   ]
012   == Call self again with the remaining arguments.
013   == Upon reaching this point, the rest
014   == of the macro is gone, so this causes a loop without
015   == any recursion.
016   %0% %3 TO *%
017 ]
```

5 Initiating and Communicating With Processes

TACL provides several ways to initiate, control, and handle the results of processes, whether they are application programs, utilities, or other TACL processes. For example, you can:

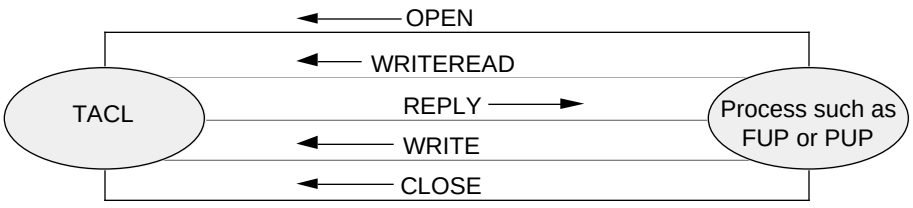
- Run a program and wait to process TACL commands until the program is finished running.
- Run a program and continue to process TACL commands while the program is running.
- Start a utility, such as FUP, and send it a list of commands.
- Start a utility, send it commands, and make decisions about further commands, depending on information obtained from the utility.

If you access a utility or other process frequently during interactive work, you can save process startup overhead by starting the process once, sending it commands as needed, and letting it run in the background until you are finished.

The following subsections describe mechanisms for starting and communicating with processes, including:

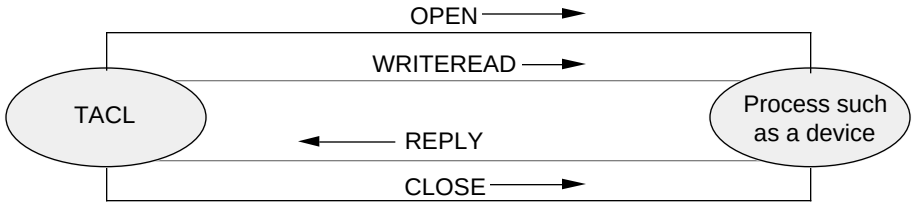
- Initiating a process: RUN and #NEWPROCESS
- Communicating with a process and retrieving results through TACL IN and OUT files, using:
  - IN and OUT files
  - INV and OUTV variables
  - The INLINE facility
  - $RECEIVE
  - #SERVER

  These methods use the following type of communication:



- ☐ Communicating with a process through $RECEIVE, using #REQUESTER, #APPEND(V), and #EXTRACT(V).

  This method uses the following type of communication:



- ☐ Processing completion information

You can use the Define Process (DP) facility (not part of TACL) to communicate with processes if your system has ViewPoint software. See "Using DefineProcess," later in this section.

Section 6, "Running TACL as a Server," describes how to access TACL as a server process.

Initiating a Process

The RUN command and #NEWPROCESS built-in function allow you to start a process. For example, the following statement starts FUP:

```
#NEWPROCESS $SYSTEM.SYSTEM.FUP /CPU 3, NAME $FUP/
```

Using RUN and #NEWPROCESS Options

TACL provides RUN and #NEWPROCESS options that allow you to specify whether your TACL process waits for the new process to finish or not. In addition, TACL provides commands and a RUN and #NEWPROCESS option (IN) that send startup information to the new process. Table 5-1 lists these options and commands.

Table 5-1. RUN and #NEWPROCESS Communication Options

Communication Requirements at Startup Time	Associated RUN or #NEWPROCESS Options and TACL Commands
Start a process and wait for it to finish; no communication	None
Start a process and return to TACL immediately; no communication	NOWAIT option
Start a process and pass information at startup time	ASSIGN, PARAM, and DEFINE commands; IN option

TACL also provides RUN and #NEWPROCESS options that support communication to the new process. For information about these options and other related functions, see "Communicating With a Process," later in this section.

The SPI and EMS interfaces provide programmatic interfaces to processes. For additional information, see Section 7, "Using Programmatic Interfaces."

Sending Information at Initiation Time

You can supply IN and OUT files for processes that accept input from an IN file and write to an OUT file. This mechanism does not allow you to evaluate the results of each request and make decisions before the next request; it does, however, provide a way to send a set of requests to a process.

In addition to supplying IN and OUT files, the following three commands send information to a new process:

- ASSIGN associates a physical file name with a logical file name.
- PARAM associates parameter values with parameter names.
- DEFINE allows you to specify a named set of attributes and values for a process.

TACL stores the associated values until you log off or until you delete the association explicitly.

The following paragraphs provide an overview of the use of these three commands. The *TACL Reference Manual* contains additional information about each command.

Assigning File Names

The ASSIGN command allows you to pass file names and, optionally, file characteristics to programs. ASSIGN associates a physical file name with a logical file name. The physical file name is any valid file name. The logical file name is defined and used within the program you are starting.

The following code assigns the file DATAFILE to the logical file name FT002 for use by a FORTRAN program, `forcalc`, that accesses the logical file FT002.

```
14> ASSIGN FT002, datafile
15> forcalc
```

TACL stores the assigned values and sends those values to requesting processes in the form of assign messages. TACL does not interpret the assigned values—that task must be performed by the application program.

A new process must request its ASSIGN messages (if any) following receipt of the startup message. The COBOL and FORTRAN compilers provide the code for this function. TAL programs that use ASSIGN commands must provide their own code for handling ASSIGN messages.

The LOGOFF command deletes existing assignments.

Defining Parameter Values

PARAM allows you to pass parameter values to a process. PARAM associates an ASCII value with a parameter name. The parameter name is an identifier that is defined and used within the program you are starting.

Use the following code to set a parameter named DEVICE_TYPE to the value 2 prior to running a program called `runit`:

```
15> PARAM DEVICE_TYPE 2
16> runit
```

TACL stores the values of parameters assigned by the PARAM command and sends these values to processes that request parameter values when the processes are started. Processes that request the parameter values interpret the values.

CLEAR PARAM *param-name* clears a specific parameter value; CLEAR ALL PARAM clears all parameter values. TACL clears all parameter values when you use the LOGOFF command.

Using DEFINEs

DEFINEs allow you to specify a named set of attributes and values for a process. There are several types of DEFINEs, many of which are intended for use with specific subsystems:

- A MAP DEFINE allows you to substitute a logical DEFINE name for an actual file name.
- A TAPE DEFINE allows you to specify labeled-tape attributes for a subsequent tape operation.
- A SPOOL DEFINE allows you to set parameters for a spooler job.
- The DEFAULTS DEFINE contains standard default values such as your default volume and subvolume name.
- The SORT and SUBSORT DEFINEs allow you to specify parameters for the FASTSORT program.

You can use the #DEFINESAVE and #DEFINERESTORE built-in functions to save a copy of one or more DEFINEs for later use. This operation is similar to a #FRAME and #UNFRAME for a set of variables; you can save the current set of DEFINEs and restore them when you are finished with your work.

For more information about how to create and use DEFINEs, see the *TACL Reference Manual* or the *Guardian User's Guide*.

Communicating With a Process

To communicate with a process, you must provide a communication path to the process. The type of path depends on how the process communicates with other processes. For example, some processes, such as FUP and PERUSE, use IN and OUT files. Other processes, such as Pathway requesters, use $RECEIVE, write to their home terminal, or use INV and OUTV variables.

The type of path also depends on whether the process opens your TACL process or your TACL process opens the other process.

If your work involves communicating with subsystems and utilities that must communicate with a terminal, the processes probably use IN and OUT files. Your TACL program can simulate a terminal if you follow these steps:

1. Start the process. The process opens your TACL process and issues a WRITEREAD operation.
2. Wait for a prompt from the process (from the WRITEREAD operation).

3. Send a command.
4. Wait for output from the process (a WRITE operation).
5. Based on the reply, decide upon the next action and either go back to Step 2 or continue to Step 6.
6. When finished, send an exit message to the process.

The following facilities support this type of operation.

- ☐ INLINE
- ☐ INV and OUTV (sometimes called implicit servers)
- ☐ #SERVER
- ☐ Define Process (part of ViewPoint)

A second communication method uses $RECEIVE to communicate with processes, devices, and files, following these steps:

1. Open the process prior to sending a request.
2. Send a request to the process.
3. Receive a reply from the process.
4. Based on the reply, decide upon the next action and either go back to Step 2 or continue to Step 5.
5. When finished, close the process.

The #REQUESTER, #APPEND(V), #EXTRACT(V), and #WAIT built-in functions support this type of communication.

There is also a third communication method that allows TACL to run as a server process. Section 6 , “Running TACL as a Server,” describes this method.

The following subsections describe these communication mechanisms and describe how to send a request, retrieve results, handle errors, and terminate communication.

Note If TACL attempts to open a process for communication at the same time that process is attempting to open TACL for communication, a deadlock condition can occur. Coordinate your use of communication mechanisms to avoid simultaneous open operations.

TACL cannot intercept messages from processes that are in block mode.

Using the INLINE Facility

The INLINE facility allows you to incorporate command stream processing into your TACL program. The syntax closely resembles interactive syntax. The INLINE facility provides the flexibility of an interactive interface; you can read process output, examine it, and make decisions about further commands.

The INLINE facility allows you to switch output variables during the operation of your TACL program. This ability can be especially useful if you are gathering information and then sending commands—such as in a PERUSE session where you obtain a list of jobs and then delete the jobs. You can work from the first set of output and use a second output variable for results and errors.

The INLINE facility supports only one active communication path at a time. You can push and define new INLINE processes while maintaining existing INLINE processes, but you can only access the most recently defined process. To communicate with more than one process at the same time, use the INV and OUTV options or #SERVER.

To use the INLINE facility, you define a prefix for commands that you send to the new process. You start the process with a RUN or #NEWPROCESS command and specify the INLINE option. Your commands appear as they would in an interactive session, except that they start with the defined prefix and a space character.

You can capture the output into a queue, examine the contents of the queue, and enable and disable output to the queue.

TACL IN and OUT files coexist with INLINE input and output streams; when you start a process with the INLINE option, TACL does not use its own IN and OUT files as the default files for the process; instead, it uses the file $*name*.#S*n*, where $*name* is the name of the TACL process and *n* is the ASCII representation of a decimal number chosen by TACL. Process I/O is therefore directed to TACL itself, which can then handle the I/O as determined by your TACL code.

Note Your TACL process must be started with a process name if you want to use the INLINE option—the operating system does not allow an unnamed process to be opened using a qualifying name, and TACL uses a qualifying name (of its own choosing) to recognize OPEN operations from processes using the INLINE option.

Table 5-2 lists the minimum set of commands or variables you use to run a process and communicate with it through the INLINE facility.

Table 5-2. INLINE Commands and Variables

Command or Function	Description
RUN or #NEWPROCESS with the INLINE option	Starts a process
#INLINEPREFIX or INLPREFIX	Sets the prefix for commands to the process
TACL command or function, preceded by the prefix and a space	Sends a request to the process
#INLINEEOF or INLEOF	Closes the process

To retrieve the current settings of the INLINE variables, expand the variables (using square brackets) or use the ENV command. The default value for #INLINEPREFIX is NULL.

Generating Input

To define the inline prefix, use the #INLINEPREFIX built-in variable or the INLPREFIX command. The following command sets the prefix to +:

```
#SET #INLINEPREFIX +
```

When TACL passes a prefixed line to a process, it first removes the prefix and the space. If a prefixed line contains square brackets, TACL evaluates them before sending the line. TACL strips comments in == (double equal sign) or { } (enclosing braces) format from prefixed lines. If a prefixed line contains only the prefix, TACL sends a blank line to the process.

If an unprefixed line generates a prefixed line, TACL passes the prefixed line to the process as its input.

In addition to program-defined termination commands, you can use the #INLINEEOF built-in function to pass an EOF to an inline process.

Lines sent as input to processes started with the INLINE option are not copied to the TACL OUT file unless you set #INLINEECHO to a nonzero value.

Use the macro in Figure 5-1, `inline_fup`, to set the INLINE prefix, start a FUP process, and send commands to FUP. Input comes from the TACL macro; output goes to the home terminal. To run this macro, load the associated file and enter:

```
12> inline_fup
```

The macro displays the FUP banner, the two commands, and FUP responses:

```
2> LOAD /KEEP 1/ INLINEEX

Loaded from $DATA.TEST.INLINEEX:
INLINE_FUP

3> inline_fup
File Utility Program - T9074C31 - (12FEB92)      System \NY
Copyright Tandem Computers Incorporated 1981,1983,1985-1992

info taclcstm
           CODE  EOF  LAST MODIF    OWNER RWEP TYPE REC BLOCK
$DATA.TEST
 TACLCSTM  101   76  11JAN92 20:43  167,1 NO-O

info mykeys
           CODE  EOF  LAST MODIF    OWNER RWEP TYPE REC BLOCK
$DATA.TEST
 MYKEYS    101  1306  1JUN91 15:19  167,1 NO-O
exit
```

Figure 5-1. Communicating With FUP

```
?SECTION inline_fup MACRO
#FRAME
#PUSH #INLINEPREFIX     == Create a new level of
                        == #INLINEPREFIX
#SET #INLINEPREFIX +    == Set the prefix to "+"
FUP /INLINE/    == Start FUP with the INLINE option
+ info taclcstm == Send FUP a command
+ info mykeys   == Send FUP another command
+ exit  == Terminate the FUP process
#UNFRAME
```

Use the macro in Figure 5-2, `script`, to build a script definition for use by a FUP process, using double slash characters (/ /) as the prefix. The syntax is:

```
script
```

When you invoke this macro, it displays the FUP banner, the SECURE command, and FUP results.

Figure 5-2. Building a Script

```
?SECTION script MACRO
#FRAME
[#DEF getinfo TEXT |BODY|   == Define the script:
// SECURE *,"NUNU"          ==   Resecure all files
// EXIT ==   Finished
]
#PUSH #INLINEPREFIX   == Save the current prefix
INLPREFIX //          == Set the new prefix
FUP/INLINE/           == Start FUP in INLINE mode
getinfo               == Input the script as if typed here
#POP #INLINEPREFIX    == Restore the previous prefix
#UNFRAME
```

Processing and Displaying Messages

The commands and variables in Table 5-3 control the display of process input and output and the capture of process output. To retrieve the setting of a built-in variable, expand the variable or use the associated command. You can set these values at any time while communicating with an INLINE process.

Table 5-3. Variables and Commands for INLINE Display

TACL Built-In Variable	TACL Command	Default Value	Description
#INLINEECHO	INLECHO	OFF	Enables or disables input line echoing to the TACL OUT file
#INLINEOUT	INLOUT	ON	Enables or disables output lines to the TACL OUT file
#INLINETO	INLTO	NULL	Specifies an output variable

Lines written by a process started with the INLINE option, and without the OUT or OUTV options, are copied to the current TACL OUT file unless the #INLINEOUT built-in variable is set to zero to disable copying.

If the #INLINETO built-in variable contains the name of a variable, TACL appends all output from processes started with the INLINE option—and without the OUT or OUTV options—to the specified variable. If #INLINETO is empty, TACL does not append the output to any variable.

The settings of #INLINEOUT and #INLINETO are not related to each other. You can send process output to the current TACL OUT file, to a variable, to both, or to neither. You can change the setting of #INLINETO to distribute inline process output among multiple variables.

Use the routine in Figure 5-3, `inline_fup_log`, to interact with FUP and send output to a variable, `log`, for FUP output. The routine displays the entire output variable at the end. The routine accepts two file name arguments and retrieves information about each file. To use this routine, load the associated file and enter:

```
inline_fup_log file1 file2
```

Figure 5-3. Retrieving Output from FUP

```
?SECTION inline_fup_log ROUTINE
#FRAME

#PUSH file1 file2
       #INLINEPREFIX   == Create a new level of #INLINEPREFIX
       #INLINETO       == Create a new level of #INLINETO
       log             == Create a variable to contain output

#SET #INLINEPREFIX +  == Set the new prefix to "+"
#SET #INLINETO log    == Associate log with process output

SINK [#ARGUMENT /VALUE file1/ FILENAME]
SINK [#ARGUMENT /VALUE file2/ FILENAME]

FUP /INLINE/          == Start FUP with the INLINE option
+ info [file1]        == Send a command to FUP
+ info [file2]        == Send a second command to FUP
#INLINEEOF            == Stop FUP
INLTO                 == Stop logging FUP output

== Display the results
#OUTPUT Here are the contents of the output log:
#OUTPUTV log
#OUTPUT End of the log.
#UNFRAME
```

To control the output displayed by TACL, use the INLOUT command. To inhibit terminal output from the INLINE_FUP_LOG routine in Figure 5-3, add an #INLINEOUT call as shown in Figure 5-4. The syntax is:

```
inline_fup_log2 file1 file2
```

Figure 5-4. Omitting Terminal Output

```
?SECTION inline_fup_log2 ROUTINE
#FRAME

#PUSH file1 file2
#PUSH #INLINEPREFIX   == Create a new level of #INLINEPREFIX
#PUSH #INLINETO       == Create a new level of #INLINETO
#PUSH log             == Create a variable to contain output

#SET #INLINEPREFIX +  == Set the new prefix to "+"
#SET #INLINETO log    == Associate log with process output

SINK [#ARGUMENT /VALUE file1/ FILENAME]
SINK [#ARGUMENT /VALUE file2/ FILENAME]

#SET #INLINEOUT 0     == Disable output to the terminal

FUP /INLINE/
+ info [file1]        == Send a command to FUP
+ info [file2]        == Send a second command to FUP
#INLINEEOF            == Stop FUP
INLTO                 == Stop logging FUP output
#SET #INLINEOUT 1

== Display the results
#OUTPUT Here are the contents of the output log:
#OUTPUTV log
#OUTPUT End of the log.
#UNFRAME
```

Use the macro in Figure 5-5, `emptyspool`, to interact with PERUSE and send output to a variable.

△ **Caution** This macro deletes jobs in the spooler. Before running it, make sure you want to delete all jobs associated with your user ID.

The macro examines each line of output and, for JOB lines, deletes the associated job. TACL returns the entire result string; you can use string-handling and character-handling functions as needed to evaluate the output and determine the next step. The syntax is:

```
emptyspool
```

The macro displays the PERUSE banner and spooled jobs; it then deletes the jobs and exits.

Figure 5-5. Deleting PERUSE Jobs

```
?SECTION emptyspool MACRO
#FRAME                == Arrange for cleanup
#PUSH #INLINEPREFIX   == Save current prefix
#PUSH #INLINETO       == Save current INLINETO
#PUSH line            == Variable for line at a time
#PUSH jobno           == Variable for current job number
#PUSH queue           == Variable for output queue
#PUSH del_rslts       == Variable for deletion results

#SET #INLPREFIX +           == Set prefix wanted here
#SET #INLINETO queue  == Use variable to collect summary

PERUSE /INLINE/       == Start PERUSE in inline mode
== The preceding command waits until the PERUSE summary is
== available and PERUSE prompts for its first command

#SET #INLINETO del_rslts == Now store deletion results

== Loop over remaining lines
[#LOOP |WHILE| NOT [#EMPTYV queue] |DO|
  #EXTRACTV queue line
  [#IF [#MATCH JOB [line]] |THEN|
    [#LOOP |WHILE| NOT [#EMPTYV queue] |DO|
      #EXTRACTV queue line        == Get the next job line
      #SETMANY jobno, [line]
      + J [jobno];DEL             == Delete the job
    ]
  ]
]
+ E                    == Tell PERUSE to exit
#UNFRAME
```

Stopping an INLINE Process

An INLINE process remains in existence until one of the following conditions occurs:

- ☐ The originating process sends an exit message to the process. The exit message can be defined by the process (such as EXIT for FUP) or can be INLEOF or #INLINEEOF.
- ☐ You log off.
- ☐ The process finishes its work and stops its own execution.
- ☐ The originating process terminates.

The following code stops an INLINE process as part of the _BREAK portion of an exception handler:

```
|_BREAK|
  #OUTPUT BREAK was pressed...terminating processing
  #PUSH x
  #SET x [#INLINEPROCESS]     == get the INLINE process name
  [#IF NOT [#EMPTYV x] |THEN|
    #INLINEEOF
  ]
  #POP x
  #RESET FRAMES
  #RETURN
```

Limitations of the INLINE Facility

Not all processes can interface with the INLINE facility. The following processes cannot interface with INLINE:

- Processes that use their home terminal (rather that IN and OUT) for their I/O; therefore, their I/O cannot be intercepted by the INLINE facility.
- Processes that do not accept IN, OUT, INV, or OUTV options.
- Processes that use terminal block mode.

Some processes behave differently when run under the INLINE facility than when run from a terminal, because some processes, when controlled by another process, change their error handling, their output format, or the rules for some of their commands.

The output from some processes changes with the release of new Tandem software. The input syntax and output formats can vary from release to release. If you process text strings, reevaluate process-dependent code for new releases of software.

The INLINE option cannot be combined with the IN or INV options. The INLINE option includes the effect of the NOWAIT option.

TACL allows more than one inline process to exist at a time, but you can communicate with only the most recent one. You must delete the most recent INLINE environment before you can communicate with its predecessor. To determine the name of the current inline process, check #INLINEPROCESS. To set up an INLINE environment, push the #INLINEPROCESS built-in variable. To delete the environment, issue an #INLINEEOF and then pop #INLINEPROCESS (or unframe it).

The ability to push and pop #INLINEPROCESS allows you to write code that uses the INLINE facility without regard to whether a program that calls your TACL code is already using the facility. You can also use this mechanism to access multiple processes within a single TACL program.

Using INV and OUTV

The INV and OUTV options for the RUN command and the #NEWPROCESS built-in function allow you to use variables in place of IN and OUT files. To use INV or OUTV, your TACL process must be a named process. You cannot use IN and INV together for the same process; nor can you use OUT and OUTV together. You can, however, start more than one process with different sets of INV and OUTV variables to communicate with more than one process.

When you use INV, the contents of the associated variable are passed line by line to the process as the process reads from its input method. There are two ways to use INV:

- Static (default): Provides a batch-type environment. Set the variable to the desired contents and start the process. TACL sends the lines one-by-one to the process. This mechanism acts much like a file. When all of the lines in the variable have been sent, subsequent reads get an end-of-file indication. You cannot add lines to the variable or change the variable after the process has started; if you attempt to add lines, TACL returns an error.
- Dynamic: Provides an interactive-type environment. You can add lines at any time during the life of the process. To specify dynamic, include the word DYNAMIC after the variable name in the INV option. TACL sends the lines one-by-one to the process. If the variable is empty, the process waits until the variable contains data.

  If you use a dynamic INV variable, you can use the PROMPT option to capture prompts. TACL places the most recent prompt string from the process into this variable.

To specify an INV or OUTV variable, start the process with a RUN or #NEWPROCESS command and include INV or OUTV and associated variables. To send a command, append the command to the variable associated with INV. To retrieve output, extract lines from the variable associated with OUTV.

When the process sends a WRITEREAD, TACL stores the prompt in the PROMPT variable (if requested with the INV PROMPT option), removes the first line of the IN variable, and passes it to the process. If the IN variable is empty, the process waits until you put data into the IN variable.

When the process writes to your program, TACL appends the line to the end of the OUT variable associated with the process. You can capture the output into a queue, examine the contents of the queue, and enable and disable output to the queue.

Table 5-4 lists the functions and options that support communication through INV and OUTV. You can request waited or nowaited communication.

Table 5-4. Functions and Options Used With INV and OUTV

TACL Command or Function	Description
RUN or #NEWPROCESS with INV and OUTV variables specified	Starts a process.
#APPEND(V) to INV variable	Sends a request to the process.
#EXTRACT(V) from OUTV variable	Retrieves a result.
#WAIT	Waits until a variable is empty (read) or full (write).
Set INV variable to #EOF or send an exit message appropriate for the process	Closes the process.

Determining the Name of the IN File

TACL assigns a name to the IN file used by the new process. The name consists of the name of your TACL process, followed by a period, a number sign, the letter S, and a number in the range 0 through 32767 ($T127.#S3, for example). Use the routine in Figure 5-6, `getname`, to start a FUP process and retrieve the name of its IN file. To use the routine, load the associated file and enter:

```
getname
```

The following session shows sample output:

```
12> getname
The IN file name is: $L11.#S9.
```

Figure 5-6. Retrieving the TACL IN File Name

```
?SECTION getname ROUTINE
#FRAME
#PUSH infile_name in_var out_var prompt_var status_var
FUP /INV in_var DYNAMIC PROMPT prompt_var, &
     OUTV out_var, STATUS status_var, NOWAIT/
SINK [#WAIT prompt_var]
#SET infile_name [#VARIABLEINFO /SERVER/ in_var]
#OUTPUT The IN file name is: [infile_name].
#UNFRAME
```

Generating Commands

Because data transfer occurs through variables, be careful when moving data to and from the variables. For example, when a process is running in the nowait mode, it is possible to attempt an operation before the preceding operation has finished.

△ **Caution** Data can be unexpectedly lost or duplicated during data transfer. To avoid lost or duplicated data, follow these guidelines.

- Use #APPEND or #APPENDV to manage IN variables. Each of these functions appends a line to a variable level in such a way that no data is lost or removed. Do not expect to examine data you have put into an IN variable—the data can be sent as soon as you append it to the variable. (If you use #SET instead of #APPEND to manage IN variables, queued lines in the IN variable can be lost before they are sent.)
- Use #EXTRACT or #EXTRACTV to manage OUT variables. Each of these functions removes the first line from a variable level in such a way that no data is lost if the process is simultaneously adding lines to a variable level. If you use #OUTPUTV to display an OUT variable and then use #SET to clear it, a new line could arrive between the #OUTPUT and the #SET; that line would be lost.
- Use #WAIT to delay processing until one of a list of variable levels is ready. #WAIT returns the fully qualified name of that variable level.
  - The OUT and PROMPT variables are ready if they have data in them.
  - An IN variable is ready if it is empty and a process is waiting for you to put data into it.
  - A STATUS variable is ready if its process was deleted.
  - Any other variable is always ready.
- Use #EOF to cause an end-of-file on an IN variable. If the IN variable is empty and a process is waiting, an end-of-file is immediately sent to the process. If the IN variable is empty and a process is not waiting, a flag is set. The next time the process attempts to read from the empty IN variable, TACL sends an EOF and the flag is cleared. #EOF does not alter the state of the process or the IN variable, but it can cause the process to terminate.

The following examples illustrate the use of INV and OUTV.

Use the macro in Figure 5-7, `fupin`, to communicate with FUP using OUTV and a static INV.

Figure 5-7. Communicating With FUP Using INV and OUTV

```
?SECTION fupin MACRO
#FRAME
PUSH in_variable        == Create an IN variable
PUSH out_variable       == Create an OUT variable

== Place a FUP command into the IN variable
#SET in_variable INFO STEIN.BOOK

FUP /INV in_variable, OUTV out_variable/

#OUTPUTV out_variable     == Display FUP output
#UNFRAME
```

You can start a process in the background and send it commands from your terminal. The following steps illustrate an interactive session with FUP:

1. Define the variables:

   ```
   9> #PUSH input_queue output_queue prompt_string
   termination_results
   ```

2. Start a FUP process that runs concurrently with the TACL process:

   ```
   10> FUP /INV input_queue DYNAMIC PROMPT prompt_string,
   OUTV output_queue, NOWAIT, STATUS termination_results/
   ```

3. Use #WAIT to make sure the previous operation has finished:

   ```
   11> SINK [#WAIT prompt_string]
   ```

4. Clear the prompt variable:

   ```
   12> #SET prompt_string
   ```

5. You can then send commands to FUP:

   ```
   13> #APPEND input_queue purge X!
   14> SINK [#WAIT prompt_string]
   15> #SET prompt_string
   16> #APPEND input_queue files
   17> SINK [#WAIT prompt_string]
   18> #SET prompt_string
   ```

6. Finally, terminate the FUP process and delete associated variables:

   ```
   19> #APPEND input_queue exit
   20> #POP input_queue output_queue prompt_string &
   termination_results
   ```

When used for productive work, include error checking for the #WAIT operations.

Use the macro in Figure 5-8, fup2, to modify a FUP operation so that user requests and FUP responses are recorded in a log file. The syntax is:

```
fup2 fup-command
```

where *fup-command* is a valid FUP command. The macro logs the request, the time of the request, and the FUP output to a file called LOGFILE. The macro displays the contents of LOGFILE before exiting.

Figure 5-8. Directing FUP Output to a Log File

```
?SECTION fup2 MACRO
#FRAME
#PUSH errvar,fupout,logvar,result  == Create variables

== Open logfile for writing
SINK [#REQUESTER /WAIT/ WRITE logfile errvar logvar]
#APPEND logvar &
  [#CONTIME [#TIMESTAMP]]: FUP %*%  FROM TERMINAL [#MYTERM]

== Write a log record
$SYSTEM.SYSTEM.FUP /OUTV fupout/ %*%      == Invoke FUP
[#LOOP |WHILE| NOT [#EMPTYV fupout]  |DO|
  == Retrieve FUP output
  [#SET result [#EXTRACT fupout] ]
  #APPEND logvar [result]    == Write FUP result to logfile
]

SINK [#REQUESTER CLOSE logvar]            == Close logfile
#OUTPUT The log contains:
FUP COPY logfile
#UNFRAME                                  == Remove variables
```

Use the macro in Figure 5-9, show_spooler_jobs, to interact with the PERUSE utility and display a list of jobs without PERUSE banner lines. To use this macro, load the associated file and enter:

```
show_spooler_jobs
```

Figure 5-9. Displaying PERUSE Jobs

```
?SECTION show_spooler_jobs MACRO
#FRAME

#PUSH line job input_queue prompt_string output_queue

== Start PERUSE as an implicit server:
PERUSE / INV input_queue DYNAMIC PROMPT prompt_string, &
     NOWAIT, OUTV output_queue/

== Wait for PERUSE to start:
SINK [#WAIT prompt_string]

== Look for a line that contains "Job":
[#IF [#LINEFINDV output_queue 1 "Job"] > 0 |THEN|
  [VDELETE /QUIET/ output_queue 1/
    [#LINEFINDV output_queue 1 "Job"] ]
]

[#LOOP |DO|
  == Retrieve one line from output_queue
  #EXTRACTV output_queue line
  #OUTPUTV line
|UNTIL|
  [#EMPTYV output_queue]
]
#EOF input_queue
#UNFRAME
```

Stopping a Process That uses INV and OUTV

A process that uses INV and OUTV remains in existence until one of the following conditions occurs:

- The originating process sends an exit message to the process. The exit message can be defined by the process (such as EXIT for FUP). Some processes accept an end-of-file indicator as a termination request. For example, an EDIT process started by the command:

  ```
  EDIT /INV in_var DYNAMIC/
  ```

  stops as soon as it reads an end-of-file produced by

  ```
  #EOF in_var
  ```

  In other cases, you can use #SERVER with the KILL option.

- You log off.
- The process finishes its work and stops its own execution.
- The originating process terminates.

TACL supports a STATUS option that stores an indication of why the process terminated. The possible indications are STOP, ABEND, CPU (CPU failure), and NET (network failure).

Limitations on the Use of INV and OUTV

Each process is associated with three to four variables. A particular variable level can be associated with only one process at a time.

A TACL process can have a maximum of 100 simultaneous openers, including active processes using variables for STATUS, PROMPT, INV, OUTV, or #REQUESTER. Multiple openers of a given process must use the process with care: If one opener has issued a WRITEREAD and is waiting for data to be appended to the IN variable of the process, any other openers attempting to do likewise receive a file system error 28.

You can specify the NOWAIT option with INV and OUTV. You must specify the NOWAIT option if you wish to examine the OUTV, PROMPT, or STATUS variables while the process is running, or if you use DYNAMIC INV.

If your TACL process ends, processes having it open receive a file-system error 66 (device downed) on all further operations after it is deleted.

If you delete a variable level (with #POP, #UNFRAME, or #KEEP) that is being used by a process as an INV, OUTV, PROMPT, or STATUS variable, TACL continues to send the contents to the process, but you can no longer access the variable level from TACL. If you use the variable as a DYNAMIC INV variable, any further references to the variable by the process receive a file system error 66 once all data has been transferred.

Using $RECEIVE

To establish your TACL program as a requester process, follow these steps:

1. If the process does not already exist, start the process as appropriate.
2. To open the process, issue a #REQUESTER READ call.
3. To send a message, append data to the prompt variable. Unlike communication to a disk file (described in Section 4, "Accessing Files"), TACL issues a WRITEREAD operation and sends the prompt data to the process or device. The process reads the data from its $RECEIVE file. When the process or device replies, TACL stores the reply into the read variable associated with the process or device.
4. To access the reply, extract lines from the read variable.
5. When finished, issue a CLOSE request and supply one of the variable levels associated with the file.

These guidelines apply to processes associated with devices as well as other types of processes.

Table 5-5 lists the functions and commands with which you can run a process and communicate through its $RECEIVE mechanism. You can request waited or nowaited I/O.

Table 5-5. Functions to Use With $RECEIVE

TACL Function or Command	Description
#REQUESTER with the READ option	Opens the process (by name)
#APPEND(V)	Sends a WRITEREAD to the process
#EXTRACT(V)	Retrieves results
#REQUESTER with the CLOSE option	Closes the process

Generating Waited and Nowaited Requests

When you communicate using $RECEIVE, you can specify waited or nowaited communication:

- ☐ For waited communication, TACL sends the WRITEREAD and continues executing code until it encounters a statement that refers to one of the #REQUESTER variables. TACL then waits until the current operation is complete before initiating the next write. To open a running process and specify waited communication, issue a #REQUESTER READ call with the WAIT option. The following statement opens $T1 for waited I/O:

```
#REQUESTER /WAIT/ READ $T1 read_error read_var prompt_var
```

- For nowaited communication, TACL sends the WRITEREAD and continues executing code. Each time you append a line to prompt_var, the TACL process reads a record from $T1 and appends it to `read_var`. To open a running process and specify nowaited communication, issue a #REQUESTER READ call without the WAIT option. The following statement opens $T1 for nowaited I/O:

  ```
  #REQUESTER READ $T1 error_var read_var prompt_var
  ```

  To wait for completion of the read operation, and when you are finished writing to the process, use the #WAIT built-in function to wait until `read_var` contains data. To avoid writing over data that has not yet been transmitted, use #WAIT to make sure the previous operation has finished.

Sending Messages to a Process

The following examples show how to use #REQUESTER to communicate with processes.

Use the routine in Figure 5-10, `send`, to open a terminal and write a message to line 25 of the terminal. The syntax is:

```
send device-id text
```

The #OUTPUT statements `output_help_text` in this example do not follow the indentation style guidelines in Section 1, "An Overview of TACL." This modification avoids extra spaces in the display.

Figure 5-10. Sending Messages to a Terminal (Page 1 of 3)

```
?SECTION send ROUTINE
[#CASE [#EXCEPTION]
|_CALL|  == code to execute when first entering the routine
  #FRAME
  [#PUSH error_message,
         destination_name,
         message,
         temp_message,
         error,
         send_message,
         req_status,
         output_time,
         origin_time,
         origin_time_hh,
         origin_time_mm]
|_ERROR|  == code to execute when an _ERROR is detected
  #ERRORTEXT /CAPTURE error_message/
  #OUTPUT Send Terminated - Error: [error_message]
  #UNFRAME
  #RETURN
```

Figure 5-10. Sending Messages to a Terminal (Page 2 of 3)

```
|_BREAK|
  #OUTPUT Send Terminated - Break Pressed
  #UNFRAME
  #RETURN
] {End CASE}

[#DEF L25 STRUCT   == Set address to line 25
  BEGIN
    BYTE BYTE (0:1) VALUE 27 111 ;
    CHAR CHAR (0:1) REDEFINES BYTE ;
  END;
]
[#DEF output_help_text TEXT
  |BODY|
  #OUTPUT/COLUMN 1/ This routine sends a message to line 25 &
of a terminal.
  #OUTPUT/COLUMN 1/ The syntax is as follows:
  #OUTPUT/COLUMN 3/ SEND terminal-name message
  #OUTPUT
  #OUTPUT/COLUMN 3/ Where: terminal-name is the &
destination terminal
  #OUTPUT/COLUMN 10/ message is the message text
  #OUTPUT
 #OUTPUT/COLUMN 3/ SEND with no arguments displays &
this information
 #OUTPUT
]
#FILTER _BREAK _ERROR

== Get the arguments.  If the user did not provide any
== arguments, or if either of the arguments is invalid,
== then display the help text.
[#CASE [#ARGUMENT/VALUE destination_name/DEVICE END
    OTHERWISE]
  |1|
    == Device name supplied.  Now get the message.
    [#CASE [#ARGUMENT /TEXT temp_message/ TEXT END]
      |1|
        == Got the text; nothing to do here
      |OTHERWISE|
        == The user did not supply a message.  Display help
        == text.
        output_help_text
        #UNFRAME
        #RETURN
    ] == end of inner #CASE
```

Figure 5-10. Sending Messages to a Terminal (Page 3 of 3)

```
    |OTHERWISE|
      == The user did not supply a device name.  Display help
      == text.
      output_help_text
      #UNFRAME
      #RETURN
    ] == end of outer #CASE
  == Obtain the current time
  == Note:  the underscores act as placeholders for unwanted
  == text.
  #SETMANY _ _ _ origin_time_hh origin_time_mm, &
    [#CONTIME [#TIMESTAMP]]
  [#IF (origin_time_mm < 10) |THEN|
    == If the minute value in the timestamp is less than 10,
    == add a zero to the front of the number to keep it at
    == two digits.
    #SET origin_time_mm 0[origin_time_mm]
  ]
  #SET origin_time [origin_time_hh]:[origin_time_mm]
  == Build the message
  #SET message [L25:char(0:1)][origin_time] == Line 25
  #SET message [message] [#USERNAME
    [#PROCESSINFO/PAID/[#MYPID]]]  == Orig userID
  #SET message [message] [temp_message]    == The message text
  == Open the destination device
  #SET req_status [#REQUESTER WRITE [destination_name] &
    error send_message]
  [#IF ( req_status <> 0 ) |THEN|
    == The open failed.  Report the error and exit.
    #OUTPUT The open of the terminal failed with Guardian &
  Error [req_status].
    #UNFRAME
    #RETURN
  ]
  #APPENDV send_message message            == Send the message
  [#IF [#MATCH [#VARIABLEINFO/VARIABLE/ &
    [#WAIT send_message error]] error] |THEN|
  #OUTPUT Write to the terminal failed:  Guardian Error [error]
  ]
  #SET req_status [#REQUESTER CLOSE error]
  [#IF ( req_status <> 0 )
   |THEN|                                 == Close failed
    #OUTPUT Close of the terminal failed:  Guardian Error &
  [req_status]
  ]
  #OUTPUT Message to [destination_name] Transmitted
  #UNFRAME
```

Testing Server Programs

You can use TACL to test server programs. One way to do this is to perform the following steps:

1. Define a request STRUCT and a reply STRUCT.
2. Verify that the server process is running.
3. Open the server using #REQUESTER with the READ option.
4. Loop and prompt for request type; use #CASE enclosures to issue specific test requests.

Communicating With $CMON

The routine in Figure 5-11, `strio`, shows how to use STRUCTs and the #REQUESTER built-in function to send messages to CMON and display replies. For more information about CMON, see the *Guardian Programming Guide*. To run this example, load the file that contains the definitions in Figure 5-11 and enter:

```
strio
```

(`strio` is listed at the end of Figure 5-11.)

Figure 5-11. Creating CMON Messages (Page 1 of 4)

```
?SECTION cmon_ci STRUCT == Common information for messages

  BEGIN
  STRUCT  groupuser;    == Group and user as separate
    BEGIN               == numbers for ease of setting
    BYTE    group;
    BYTE    user;
    END;
  INT     cipri;
  FNAME   ciinfile;
  FNAME   cioutfile;
  END;

?SECTION logon_msg STRUCT

  BEGIN
  INT     msgcode VALUE -50;
  STRUCT  ci;
    LIKE  cmon_ci;
  END;
```

Figure 5-11. Creating CMON Messages (Page 2 of 4)

```
?SECTION logon_reply STRUCT

  BEGIN
  INT     replycode;
  CHAR    replytext(0:131);
  END;

?SECTION logoff_msg STRUCT

  BEGIN
  INT     msgcode VALUE -51;
  STRUCT  ci;
    LIKE  cmon_ci;
  END;

?SECTION logoff_reply STRUCT

  BEGIN
  INT     replycode;
  CHAR    replytext(0:131);
  END;

?SECTION processcreation_msg STRUCT

  BEGIN
  INT     msgcode VALUE -52;
  STRUCT  ci;
    LIKE  cmon_ci;
  FNAME   progname;
  INT     priority;
  INT     processor;
  FNAME   proginfile;
  FNAME   progoutfile;
  FNAME   proglibfile;
  FNAME   progswapfile;
  END;

?SECTION processcreation_reply STRUCT

  BEGIN
  INT     replycode;
  CHAR    replytext(0:131);
```

Figure 5-11. Creating CMON Messages (Page 3 of 4)

```
  STRUCT  run_info REDEFINES replytext;
    BEGIN
    FNAME   progname;
    INT     priority;
    INT     processor;
    END;
  END;

?SECTION talk_to_cmon ROUTINE
== This routine accepts the names of two STRUCTs as its
== arguments.  The first has been set by the caller to the
== message to send to CMON and the second is set by this
== routine to the response from CMON.
#FRAME

== Get the names of the two structures
#PUSH to_cmon from_cmon
#IF [#ARGUMENT/VALUE to_cmon/ VARIABLE]
#IF [#ARGUMENT/VALUE from_cmon/ VARIABLE]
#IF [#ARGUMENT END]

== Set the common information for the caller
#SET [to_cmon]:ci:groupuser [grp_usr &
     [#PROCESSINFO/PAID/ [#MYPID]]]
#SET [to_cmon]:ci:cipri [#PROCESSINFO/PRI/ [#MYPID]]
#SET [to_cmon]:ci:ciinfile [#MYTERM]
#SET [to_cmon]:ci:cioutfile [#MYTERM]

== Open CMON
#PUSH e r p
#IF [#REQUESTER/WAIT 5000/ READ $cmon e r p]

== Show the message being sent
#OUTPUT -- Message to $CMON --
#OUTPUTV [to_cmon]
#OUTPUT ----------------------

== Send the message and get the reply
#APPENDV p [to_cmon]
#EXTRACTV r [from_cmon]

== Show the reply
#OUTPUT -- Reply from $CMON --
#OUTPUTV [from_cmon]
#OUTPUT ----------------------
== Exit, popping the variables and closing CMON
#UNFRAME
```

Figure 5-11. Creating CMON Messages (Page 4 of 4)

```
?SECTION grp_usr ROUTINE
== Converts a group,user to a space-separated list
#FRAME
#PUSH v
#IF [#ARGUMENT/VALUE v/ NUMBER]
#RESULT [v]
#IF [#ARGUMENT COMMA]
#IF [#ARGUMENT/VALUE v/ NUMBER]
#RESULT [v]
#UNFRAME

?SECTION strio ROUTINE
== Invoke this for the demo
#FRAME

== Do a simulated LOGON
#DEF msg STRUCT LIKE logon_msg;
#DEF reply STRUCT LIKE logon_reply;
talk_to_cmon msg reply

== Do a simulated processcreation
#DEF msg STRUCT LIKE processcreation_msg;
#SET msg:progname tacl   == These are simulated RUN options
#SET msg:priority 148
#SET msg:processor 13
#SET msg:proginfile taclin
#SET msg:progoutfile taclout
#SET msg:proglibfile tacllib
#SET msg:progswapfile taclswap
#DEF reply STRUCT LIKE processcreation_reply;
talk_to_cmon msg reply

== Do a simulated LOGOFF
#DEF msg STRUCT LIKE logoff_msg;
#DEF reply STRUCT LIKE logoff_reply;
talk_to_cmon msg reply

#UNFRAME
```

Stopping a Process That Communicates Using $RECEIVE

Your ability to stop this type of process depends on how the process started and your access capabilities. The process might stop if one of the following occurs:

- ☐ The process finishes its work and stops its own execution.
- ☐ The originating process terminates.
- ☐ The originating process sends an end-of-file or other end signal (for example, exit for FUP).

Using #SERVER

If you want your TACL program to communicate with more than one process at once, you can define subdevice names for it. This qualified process name is known as a server path. Processes can use the server path to open your TACL process by name. This mechanism is very flexible; it can be used for complex processing such as multiple input streams.

Your TACL process must have a process name for you to be able to use #SERVER.

For information about accessing a TACL process as a server, see Section 6, “Running TACL as a Server.”

Naming Your Process

To create a server path for your TACL process, use the #SERVER built-in function. After TACL creates the name, other processes can open your TACL as if it were a file so that you can write to and read from those processes. For example:

```
#PUSH server_name in_var out_var prompt_var
#SET server_name [#PROCESSINFO/PROCESSID/].#A39
SINK [#SERVER /IN in_var, PROMPT prompt_var, OUT out_var/ &
  [server_name]]
```

Valid names consist of your TACL process name followed by a period, a number sign (#), a letter, and zero to six alphanumeric characters ($T127.#A39, for example). If you want, that name can be followed by another period, a letter, and zero to seven alphanumeric characters ($T127.#A39.Z48F, for example).

If you do not specify a name, TACL supplies the name for you. The name consists of the process name of your TACL followed by a period, a number sign, the letter S, and a number in the range 0 through 32767:

```
#PUSH server_name in_var out_var prompt_var
#SET server_name [#SERVER /IN in_var, PROMPT prompt_var, &
                          OUT out_var/]
```

Sending Commands

When your TACL program has a server path, other processes can open it and issue WRITEREAD operations. TACL receives a prompt; your TACL process can then write to the other process. This code is exactly the same as that used with #REQUESTER in "Using $RECEIVE," earlier in this section.

Use the macro in Figure 5-12, `serv`, to communicate with a FUP process. The macro obtains a server name and then uses it for the IN and OUT files of the FUP process. The macro requests an INFO * operation.

Figure 5-12. Communicating With FUP Using #SERVER

```
?SECTION serv MACRO
#FRAME
#PUSH server_name in_v out_v prompt_v  response lines

== Assign a name to this process
#SET server_name [#SERVER /IN in_v, PROMPT prompt_v, OUT
out_v/]

== Start FUP and use this process for its IN and OUT files
FUP /IN [server_name], OUT [server_name], NAME $fp, NOWAIT/

== Wait for a prompt and issue a command
SINK [#WAIT prompt_v]
#APPEND in_v INFO *

SINK [#WAIT in_v]
SINK [#WAIT out_v]

== When we have a result, extract it and display all lines
#SET lines [#VARIABLEINFO /LINES/ out_v]
[#LOOP |WHILE| lines > 0 |DO|
  #EXTRACTV out_v response
  #OUTPUTV response
  #SET lines [#COMPUTE [lines] - 1]
]
STOP $fp
#UNFRAME
```

Note You could use #NEWPROCESS instead of RUN in the previous example, which would allow you to handle the results of the process initiation.

Limitations on the Use of #SERVER

TACL supports only one active READ at a time from each server path. If more than one process sends a READ or WRITEREAD to a server path, TACL process the first such request that arrives and returns a file system error 28 to each of the other processes. For additional limitations, see the *TACL Reference Manual*.

Deleting the Process Name

Your program remains accessible until you delete the name with a call to #SERVER with the KILL option. For example:

```
#SERVER /KILL/ server-name
```

If you log off, all server paths are deleted immediately.

Using Define Process

The Define Process (DP) facility is a library of TACL commands that is part of the ViewPoint product. DP provides a mechanism for starting and communicating with processes. If ViewPoint is installed on your system, you can use DP to:

- ☐ Define and start one or more background processes
- ☐ Communicate with a background process interactively or with a list of commands
- ☐ Manage response data

As with other asynchronous background processes, you can use DP to run a process and access it frequently to avoid the startup overhead associated with multiple initializations of the same program.

To use DP, add the DP directory to your uselist:

```
#SET #USELIST [#USELIST] DP
```

Next, associate a name with a background process. The following code starts a FUP process, associates the name F with it, requests a FUP INFO * operation, and then deletes the FUP process and associated variable:

```
20> DP FUP /PNAME f/
21> F INFO *

          CODE    EOF    LAST MODIF     OWNER      RWEP
$DATA.SV
 MYKEYS   101    2536  16MAY92 13:01     8,16      CUCU
 TACLCSTM 101   20480  24SEP91 15:20     8,16      CUCU
22>
...
44> UNDP F
```

For more information about DP, see the *ViewPoint Manual.*

Processing Completion Information

Depending on how you run a process and how the process handles termination, TACL can access several types of completion information:

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

Processes can be run as batch jobs under the control of the NetBatch product. The following paragraphs describe the role and use of TACL as a NetBatch command interpreter and how to monitor the status of jobs in progress. For detailed information on scheduling processes as batch jobs, see the *NetBatch Manual.*

Processing NetBatch Jobs and Completion Codes

TACL saves completion code information in the variable :_COMPLETION, if it exists. Tandem code in the standard TACLSEGF defines :_COMPLETION as follows:

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

TACL sets this variable whenever you attempt to start a process, whenever you successfully start a process, and whenever a process you started terminates (successfully or otherwise). Whenever TACL sets this variable, its previous contents are lost.

If you define a variable named :_COMPLETION, it should be a STRUCT. Each time TACL stores data in :_COMPLETION, it resets the STRUCT to default values. If the STRUCT is shorter than the data to be stored in it, the extra data is discarded. If the STRUCT is longer than the data to be stored in it, the extra space remains set to default values.

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

If a NEWPROCESS failure occurs while you are trying to start a process, TACL sets MESSAGECODE to 0, COMPLETIONCODE to 4, and TERMINATIONINFO to the NEWPROCESS error code.

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

When a process terminates (STOP or ABEND), the system message is put into :_COMPLETION.

An interactive TACL displays completion code information whenever one or more of the following are true:

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

For more information about MESSAGECODE and other definitions, see the *System Procedure Calls Manual*.

The display shows only those numeric fields that are nonzero, and displays the amount of TEXT indicated by TEXTLENGTH.

Use the macro in Figure 5-13, `sqlcomp`, to perform preparation, COBOL85, and SQL compile steps for a COBOL85 program called SRCFILE. The syntax is:

```
sqlcomp
```

Figure 5-13. Checking Completion Codes (Page 1 of 2)

```
?SECTION sqlcomp ROUTINE
#FRAME
== Purge files from the previous compilation
FUP PURGE CBLFL, SQLFL!

== Run the preprocessor as a waited process
SQLCOBOL /name,CPU 3, PRI 140, IN SRCFILE, OUT $S.#PM/CBLFL,
SQLFL
```

Figure 5-13. Checking Completion Codes (Page 2 of 2)

```
== Check the completion code; if an error occurred, display
== it and exit
[#IF [:_COMPLETION:COMPLETIONCODE] <> 0 |THEN|
  #OUTPUT *** Error during preprocessing
  #OUTPUT Completion code = [:_COMPLETION:COMPLETIONCODE]
  #UNFRAME
  #RETURN
]

== Compile and bind with COBOL85, again as a waited process
PARAM BINSERV $SYSTEM.SYSTEM.BINSERV
PARAM SYMSERV $SYSTEM.SYSTEM.SYMSERV
COBOL85 /name,CPU 3, PRI 140, IN CBLFL, OUT $S.#PM/APPOBJ

== Check the completion code; if an error occurred, display
== it and exit
[#IF [:_COMPLETION:COMPLETIONCODE] <> 0 |THEN|
  #OUTPUT *** Error during COBOL85 compilation and binding
  #OUTPUT Completion code = [:_COMPLETION:COMPLETIONCODE]
  #UNFRAME
  #RETURN
]

== SQL compilation; waited process
SQLCOMP /name,CPU 3, PRI 140, IN APPOBJ, OUT $S.#PM/

== Check the completion code; if an error occurred, display
== it and exit
[#IF [:_COMPLETION:COMPLETIONCODE] <> 0 |THEN|
#OUTPUT *** Error during SQL compilation
#OUTPUT Completion code = [:_COMPLETION:COMPLETIONCODE]
#UNFRAME
#RETURN
]
```

Monitoring Job Status: ENQUIRY

The ENQUIRY facility allows you to acquire the last 22 lines written to the OUT file of a TACL process; it provides the same functionality as the NetBatch-Plus ENQUIRY screen. To access this information, you open the TACL process and send it an enquiry message (-22). TACL replies with the lines of text from its OUT buffer.

Note To limit open access to a TACL process, use the #TACLSECURITY built-in function.

The following considerations apply:

- If a line in the buffer is less than 80 bytes long, TACL does not pad the remaining bytes. The byte length of the line can be used to determine the amount of valid information on the line.
- If a line in the buffer was greater than 80 bytes, TACL truncates the text to 80 bytes but records the actual length in the buffer.
- TACL writes a copy of all OUT data to the buffer; it does not perform any filtering or checking of the content of the data, except that it does not copy blank lines to the buffer.
- TACL initializes the buffer to nulls. If you detect a null line length, the remaining part of the buffer is empty. You can use this information to determine where in the buffer to start reading. On the initial read, if the next line field has a zero length, you must begin reading at line 0.

Use the routine in Figure 5-14, `enquiry`, to access the ENQUIRY feature of TACL. The `enquiry` routine acquires the last 22 lines that a TACL process has written to its primary OUT file. The routine performs the following steps:

1. Opens the TACL process specified by the `process_name` variable.
2. Sends the TACL process an enquiry message (message code of -22).
3. Receives a reply buffer (`logring`) containing the last 22 output lines.
4. Displays the buffer as follows:

   If the `logring` buffer is full (typical situation):

   a. Start reading `logring` at the line number specified by the `next` element of the `logring` STRUCT. When the line `last_line` is read, start reading at line 0 and terminate after reading the line `next` - 1.

   b. As each line is read, display the number of characters specified in the length element of the `logring` STRUCT.

   c. If the length exceeds `max_chars`, truncate the line to the length of `max_chars`.

   If `logring:next` points to a line with zero length, the buffer is only partially full. Start reading `logring` at line 0 and terminate after reading line `logring:next`-1.

To run this routine, load the associated file and enter:

```
enquiry TACL-process-name
```

Figure 5-14. Retrieving TACL Output (Page 1 of 2)

```
?SECTION enquiry ROUTINE
#FRAME
[#PUSH
  last_line last_char max_chars
  status r_error r_rec prompt
  process_name
  next last record length
]
#SET max_chars 80       == Truncate display at this position
#SET last_line 21
#SET last_char [#COMPUTE [max_chars] - 1]
[#DEF enquiry_message
  STRUCT
    BEGIN
    INT message_code VALUE -22;
    END;
]

[#DEF enquiry_reply
  STRUCT
    BEGIN
    INT   next;
    STRUCT line (0:[last_line]);
      BEGIN
      INT length;
      CHAR text (0:[last_char]);
      END;
    END;
]

== Get the name of the TACL process
SINK [#ARGUMENT /TEXT process_name/ PROCESSNAME]
==
==  Open the TACL process.
#SET status &
  [#REQUESTER/WAIT 5000/READ [process_name] r_error
    r_rec prompt]

==  If unsuccessful process open, display an error and exit.
[#IF [status] <> 0 |THEN|
  #OUTPUT Error opening the process:  [status]
  #UNFRAME
  #RETURN
]
```

Figure 5-14. Retrieving TACL Output (Page 2 of 2)

```
== Successful open:
== Send our message and get a reply (like a WRITEREAD)
#APPENDV prompt enquiry_message  == The WRITE
#EXTRACTV r_rec enquiry_reply    == The READ

#SET next [enquiry_reply:next]
#SET last [enquiry_reply:next]

== A zero length at this point means a partial buffer, so
== start processing at the beginning
[#IF ([enquiry_reply:line([next]):length] = 0) |THEN|
  #SET next 0
== If we do not have an empty buffer, begin the processing
[#IF ([enquiry_reply:line([next]):length] <> 0) |THEN|
  [#LOOP |DO|
    == Check the line length and truncate if necessary
    [#IF [enquiry_reply:line([next]):length] > [max_chars]
    |THEN|
      #SET enquiry_reply:line([next]):length [max_chars]
    ]

    == Display it for length characters
    #SET length [#COMPUTE   &
       [enquiry_reply:line([next]):length] - 1]
    [#OUTPUT [enquiry_reply:line([next]):text(0:[length])]]

    == Compute the position of the next read
    #SET next [#COMPUTE [next] + 1]

    == Should we wrap around to the beginning yet?
    [#IF [next] > [last_line] |THEN|
      #SET next 0
    ]

    |UNTIL| [next] = [last]
  ] == LOOP

|ELSE|
   == Nothing to display, the buffer is empty.
  #OUTPUT Buffer is empty
] == IF

==  Close process.
SINK [#REQUESTER/WAIT/CLOSE r_rec]

#UNFRAME
```

(This page left intentionally blank)

6 Running TACL as a Server

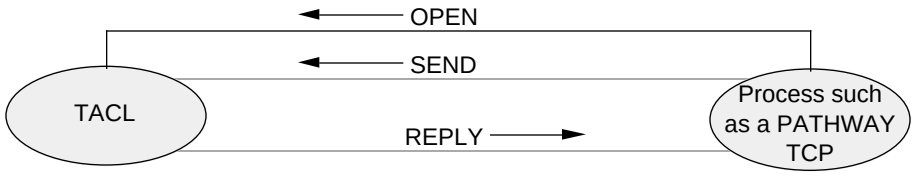
Section 5, “Initiating and Communicating With Processes,” described how to generate requests and communicate with processes from within a TACL program. This section discusses a different perspective: how to provide access to a TACL process from other processes.

There are two types of TACL servers:

- A generic TACL process that processes TACL commands and built-in functions such as WHO, ENV, CREATE, and #TIMESTAMP and returns results.
- A TACL process that interprets a program (of type TEXT, MACRO, or ROUTINE). This type of server processes a set of requests defined by the program. To provide this type of access, you can access a program from your TACLCSTM file or use the #SERVER built-in function.

The second type of server is a special form of the first type. Both processes are true servers as defined by requester-server architecture: these processes wait until they receive a request on $RECEIVE and then process the request and reply with the results.

Both of these mechanisms support the following type of communication:



The following subsections describe how to create both types of servers.

Note When operating as a server, TACL does not act as a fault-tolerant server (it does not check the sync ID). Therefore, do not use TACL to check correct operation of a fault-tolerant application.

Running a TACL Process as a Server

Whenever TACL uses $RECEIVE for input, the TACL process is considered to be a server. To run TACL as a server, you must do two things:

- Establish the TACL IN file as $RECEIVE.
- Provide a name for the TACL process so that other processes can access it.

The following paragraphs describe how to start TACL as a server, communicate with it, and manage output.

Starting TACL as a Server Process

To run TACL as a server, use the RUN or #NEWPROCESS function with TACL as the program file and IN set to $RECEIVE:

```
23> TACL /NAME $name, IN $RECEIVE, NOWAIT/
24>
```

When operating as a server, TACL starts up, logs on under its process accessor ID, and processes input in the usual way.

Sending Requests to a TACL Server

A requesting TACL process sends requests as described in Section 5, "Initiating and Communicating With Processes." Table 6-1 lists commands used to communicate with a server.

Table 6-1. Functions That Support Interprocess Communication

TACL Function	Description
#REQUESTER with the WRITE option	Opens the background TACL process
#APPEND(V)	Sends a WRITEREAD to the TACL process
#EXTRACT(V)	Retrieves results
#REQUESTER with the CLOSE option	Closes the process

Use the macro in Figure 6-1, `runsrv`, to read requests from the home terminal and send the requests to a background TACL process. The background TACL process executes each request and returns results to #MYTERM. To invoke this macro, load the file and enter:

```
runsrv
```

Figure 6-1. Starting and Sending Requests to a TACL Server (Page 1 of 2)

```
?SECTION runsrv MACRO
#FRAME
#PUSH err_var write_var temp_var rslt

== ** The following code starts the server process **
== Start the background TACL process, $mrt, if it is not
== already running.  The IN file is $RECEIVE.
[#IF NOT [#PROCESSEXISTS $mrt] |THEN|
        TACL /IN $receive, OUT [#MYTERM], NAME $mrt, NOWAIT/
]

== ** The following code accesses the TACL server **
== Open the TACL server process ($mrt)
#SET rslt [#REQUESTER /WAIT/ WRITE $mrt err_var write_var]
[#IF rslt = 0 |THEN|
  #OUTPUT $mrt opened successfully
|ELSE|
  #OUTPUT $mrt not open; file error [file1_err]
  #RETURN
]
```

Figure 6-1. Starting and Sending Requests to a TACL Server (Page 2 of 2)

```
[#LOOP |DO|
  #SET temp_var [#INPUT Enter Command:]
  [#IF [#MATCH EXIT [#SHIFTSTRING /UP/ [temp_var]]] |THEN|
    #SET #INPUTEOF -1
  |ELSE|
    == Send a command to the background TACL
    #APPENDV write_var temp_var
  ]
|UNTIL| [#INPUTEOF]
]
SINK [#REQUESTER CLOSE write_var]
#UNFRAME
```

When you run `runsrv`, it displays the following:

```
10> runsrv
Enter Command:
```

You can then enter a TACL command or built-in function. To exit, type Control-Y.

When TACL starts the background process, that TACL process logs on under its process accessor ID. The background process in Figure 6-1 accepts the standard set of TACL commands and built-in functions; to create a TACL server that runs your code, see the next subsection, “Running TACL Code as a Server.”

When TACL is operating as a server, it queues messages if they arrive while TACL is already working on a message. The depth of this queue is limited solely by the availability of space in the TACL segment. If the message cannot be queued, TACL responds with file-system error 45 (file is full).

When the number of processes that have the TACL process open goes from nonzero to zero, and #INPUT(V) detects that transition, #INPUTEOF is set to a nonzero value. If the transition is detected in response to a prompt generated by TACL itself, #EXIT is set to a nonzero value. This behavior is analogous to that which occurs when end-of-file (EOF) is reached on the IN file when TACL is not operating as a server.

Special Actions of the #INPUT(V) Built-In Function

When TACL acts as a server, it can process statements from $RECEIVE or statements coded within the TACL process. Action of the #INPUT(V) built-in function depends on the origin of the #INPUT(V) call:

- When a server TACL process executes an #INPUT(V) built-in function as part of its own code, it reads from $RECEIVE, executes the text in that message, and replies with the results.
- When a requester sends a message that contains an #INPUT(V) built-in function, TACL behaves differently. Upon recognizing the #INPUT(V) built-in function, TACL replies immediately with whatever is contained within its reply buffer. It then reads the next message from $RECEIVE and postpones the processing of any remaining functions or commands that may have followed the #INPUT(V) call in the first request. After receiving the second message, TACL processes the text in that message and the resumes processing the text left over from the first message. After TACL processes all of the message text, it replies with the results of the second message and the results from the leftover text in the first message.

The second mechanism is not used frequently; the description is provided to explain the action of TACL in case such use should occur.

Directing Output From TACL

As a server, TACL can issue only one response for each request—a operating system restriction. Requests and responses are limited to 5000 bytes, although they can span multiple lines.

Specifying the OUT File

TACL normally sends output to its OUT file. When TACL operates as a server, you can specify the OUT file as $RECEIVE:

```
23> TACL /NAME $name, IN $RECEIVE, OUT $RECEIVE, NOWAIT/
```

If you specify OUT as $RECEIVE, all normal output (up to 5000 bytes) is included in the next REPLY. If you specify OUT *filename*, TACL writes all its output to *filename*, as usual, and does not include it in the next REPLY.

If you supply OUT without a file name, TACL discards its normal output.

This flexibility allows you to construct TACL environments that function properly whether TACL is operating as a requester or as a server.

Running TACL Code as a Server

To provide TACL code for use by a TACL server process, you must provide a link to a TACL process that is running your code. There are several ways to accomplish this, including:

- ☐ ASSIGN the TACLCSTM file to a TACL macro or routine that contains the code.
- ☐ Include access to the code from your TACLCSTM file. (When you start a TACL process, the process accesses your TACLCSTM file.) Your TACLCSTM file can check #TACLOPERATION, which indicates whether a TACL process is receiving input from an IN file (REQUESTER) or $RECEIVE (SERVER).
- ☐ Process the input through a server path defined by the #SERVER built-in function, described in Section 5, "Initiating and Communicating With Processes."

Open the TACL process as described in Section 5.

Constructing a TACL Server

One way to accept requests and reply with messages in a specified format is to build a loop using #INPUTV and #REPLYV and protect it with an exception handler. Figure 6-2 shows an example a TACL program that uses an #INPUTV and #REPLYV loop.

Using #REPLYPREFIX

Pathway programs can use TACL as a server. You can use #REPLYPREFIX to prefix each reply with a 16-bit binary code. The #REPLYPREFIX built-in variable can be empty or it can contain any numeric value in the range 0 through 65535. If #REPLYPREFIX is not empty, its value precedes each response; if #REPLYPREFIX is set, replies can be up to 5002 bytes long.

Using #REPLY and #REPLYV

Normally, if the OUT file is not set to $RECEIVE, then all output is excluded from the reply. However, #REPLY(V) forces text to be included in the reply buffer, regardless of the setting of the OUT file.

Requests and replies (except the part generated by #REPLYPREFIX) consist of ASCII characters with internal line breaks encoded as null bytes.

When functioning as a server, TACL stores #REPLY(V) text and, if OUT is set to $RECEIVE, its normal output as well. TACL replies with the buffer contents when it has processed all of the text that was included in the request. If the accumulated response exceeds 5000 bytes, TACL discards the excess without notification.

TACL can have as many as 100 openers. All openers of a given TACL process can never have more than one message outstanding to TACL at a time, unless the processes have opened TACL without a #SERVER name.

Using TACL as a Pathway Server

The code in this subsection creates a Pathway environment that includes one TCP and one TACL server. The TACL routine processes three types of requests:

- CREATE file
- PRINT file
- PURGE file

This example uses the following files:

- TACLIN—The TACL routine in Figure 6-2
- SCOBSRC—The Screen COBOL source for the requester in Figure 6-3
- PWYOBEY—The obey file that starts the Pathway environment, listed after Figure 6-3
- PATHCNFG—The Pathway configuration file in Figure 6-4

In addition, the Pathway configuration file uses a file named LOG1 for logging; create this file before starting the Pathway environment.

When the application runs, it presents a screen that lists the three functions. If you type a request, TACL processes the request and returns a response. The application displays this response on your screen. To clear the screen, type F1. You can then enter another request or exit the application.

Note If you use these files on your system, change the volume and subvolume names to reflect storage locations on your system.

Use the code in Figure 6-2 as a sample Pathway server. This file is known as TACLIN in the Pathway configuration file; if you use a different file name, modify the configuration file.

Figure 6-2. Running a TACL Program as a Server (Page 1 of 2)

```
?TACL ROUTINE
#FRAME

[#DEF process routine |BODY|
  #FRAME
  #PUSH param status
  [#CASE [#ARGUMENT KEYWORD /WORDLIST create/
                    KEYWORD /WORDLIST print /
                    KEYWORD /WORDLIST purge /
                    OTHERWISE
         ]
  | 1 | {CREATE file}
    [#CASE [#ARGUMENT /VALUE param/ FILENAME TEMPLATE END
      TEXT]
    | 1 |  #RESULT File [param] already exists !
    | 2 |  CREATE [param]
           #RESULT File [param] created
    | 3 |  #RESULT No file name supplied!
    | 4 |  #RESULT Invalid file name: [param]
    ]
  | 2 | {PRINT file}
    [#CASE [#ARGUMENT /VALUE param/ FILENAME END TEXT]
     | 1 | fup /NOWAIT,OUT $null/copy [param],$s.#prt
           #RESULT Printing of file [param] started
     | 2 | #RESULT No file name supplied!
     | 3 | #RESULT Invalid file name or file does not exist:&
             [param]
    ]
  | 3 | {PURGE file}
    [#CASE [#ARGUMENT /VALUE param/ FILENAME END TEXT]
    | 1 | #SET status [#PURGE [param]]
          [#IF status |THEN|
            #RESULT Error [status] during purge of file &
              [param]
          |ELSE|
            #RESULT File [param] purged
          ]
    | 2 | #RESULT No file name supplied!
    | 3 | #RESULT Invalid file name or file does not exist:&
            [param]
    ]
```

Figure 6-2. Running a TACL Program as a Server (Page 2 of 2)

```
    | 4 | {invalid request}
      #RESULT Invalid request !
      #UNFRAME
      #RETURN
    ]
  #UNFRAME
  ]

  [#DEF reply ROUTINE |BODY|   == Reply to Screen COBOL
    #FRAME
    #PUSH text
    [#DEF  answer80 STRUCT     == 80-character text response
       BEGIN
         CHAR byte(0:79);
       END;
    ]
    #IF [#ARGUMENT /VALUE text/ TEXT]
    #SET     #REPLYPREFIX 0    == constant reply code
    #SET     answer80  [#CHARGET text 1 to 80]
    [#IF [#INTERACTIVE] |THEN|
       #OUTPUT [answer80]      == Display the result
    |ELSE|                     == or
       #REPLYV  answer80       == Reply
    ]
  #UNFRAME
  ]

  ==main loop

  #PUSH request prompt outcome end_flag
  #SET end_flag 0

  [#LOOP |DO|
    == Read a record
    #INPUTV / NOECHO / request prompt
    [#IF [#INPUTEOF] |THEN|
    #SET end_flag 0
    |ELSE|
      #SET outcome [process [request] ]
      reply [outcome]
    ]
  |UNTIL| end_flag
  ]

  #UNFRAME
```

Figure 6-3 contains the Screen COBOL source code for the requester. This requester displays a screen and waits until the user presses the F1 key to send a request to TACL or SF16 to exit the program. If the user sends a request, the requester displays the response from TACL and waits until the user presses the F1 key to clear the message. The requester then prompts for input again.

To compile this program, type a statement, such as the following, with appropriate file and spooler names for your system:

```
SCOBOLX/IN SCOBSRC, OUT $S.#TCP/
```

When you compile the requester, Screen COBOL produces two files: POBJCOD and POBJDIR.

Figure 6-3. Screen COBOL Code That Accesses a TACL Server (Page 1 of 2)

```
Identification Division.
 Program-Id.
  TEST-TACL.
Environment Division.
 Configuration Section.
  Source-Computer.
   EXT.
  Object-Computer.
   EXT, Terminal Is T16-6530.
  Special-Names.
   REVERSE Reverse
   F1      F1
   SF16    Sf16.
Data Division.
 Working-Storage Section.
  1 TOTACL Pic X(80).
  1 TACL   Pic X(80).
  1 REPCOD Pic 9(4) Comp.
  1 FEOJ   Pic 9 Value 0.
    88 EXI Value 1.
 Screen Section.
   1 FRAME.
   2 Filler At 1, 31 Value "**** TACL SERVER ****" REVERSE".
   2 Filler At 7, 1 Value "ENTER REQUEST:".
   2 Filler At 8, 1 Value "CREATE file, PRINT file, or ".
   2 Filler At 9,  1 Value "PURGE file, then press F1".
   2 Filler At 10, 1 Value "-- or --  press SF16 to exit:".
   2 Filler At 11, 1 Pic X(80) To TOTACL.
   2 Filler At 15, 1 Value "TACL Response; F1 to continue:".
   2 RESPONSE At 16, 1 Pic X(80) From TACL.
```

Figure 6-3. Screen COBOL Code That Accesses a TACL Server (Page 2 of 2)

```
Procedure Division.
  0-DEBUT.
    Display Base FRAME
    Perform 1-LOOP Until EXI
    Stop Run.
  1-LOOP.
    Accept FRAME Until F1 Escape SF16
    If Termination-Status = 1
*   Then
       Send TOTACL To "TACL"
         Reply Code 0 Yields REPCOD TACL
       Display RESPONSE
       Accept Escape F1
       Reset RESPONSE
       Clear Input
    Else
       Move 1 To FEOJ.
*   End-If
```

The following commands start the Pathway environment. You can store these commands in an OBEY file:

```
PATHMON /NAME $tst, NOWAIT/
PATHCOM /IN pathcnfg/$tst
```

The commands in Figure 6-4 configure the sample Pathway environment. This file is used as input when you start the Pathway environment. Note that the server program file is $SYSTEM.SYS01.TACL and the commands assign the TACLCSTM file to the TACLIN file.

This example uses $DATA for the volume and TEST for subvolume; modify these entries to reflect appropriate volume and subvolume names on your system.

Figure 6-4. Configuring the Pathway Environment

```
SET PATHMON BACKUPCPU 1
SET PATHWAY MAXASSIGNS 1
SET PATHWAY MAXDEFINES 1
SET PATHWAY MAXPARAMS 1
SET PATHWAY MAXPATHCOMS 1
SET PATHWAY MAXPROGRAMS 1
SET PATHWAY MAXSERVERCLASSES 1
SET PATHWAY MAXSERVERPROCESSES 5
SET PATHWAY MAXSTARTUPS 1
SET PATHWAY MAXTCPS 1
SET PATHWAY MAXTERMS 10
START PATHWAY COLD !

RESET TCP
SET TCP MAXREPLY 100
SET TCP MAXTERMS 10
SET TCP PROGRAM $SYSTEM.SYSTEM.PATHTCP2
SET TCP TCLPROG $DATA.TEST.POBJ
SET TCP CPUS 2:1
ADD TCP TCP-1

RESET PROGRAM
  SET PROGRAM TMF ON
  SET PROGRAM TYPE T16-6530 (INITIAL TEST-TACL )
  SET PROGRAM TCP TCP-1
ADD PROGRAM TEST-TACL

RESET SERVER
  SET SERVER CPUS (1:2)
  SET SERVER MAXSERVERS 2
  SET SERVER NUMSTATIC 1
  SET SERVER PROGRAM $SYSTEM.SYS01.TACL
  SET SERVER TMF OFF
  SET SERVER ASSIGN TACLCSTM, $DATA.TEST.TACLIN
  SET SERVER IN $RECEIVE
  SET SERVER OUT $NULL
ADD SERVER TACL
START TCP *
START SERVER *
LOG1 $DATA.TEST.LOG1
```

To run this application, type the following:

```
12> O PWYOBEY
(Pathway displays startup information here)

13> PATHCOM $TST
=RUN TEST-TACL
(Pathway displays the application screen.  When finished,
press SF16 instead of entering a request.)

=SHUTDOWN, WAIT
TCP TCP-1, STOPPED
=EXIT
14>
```

Press F1 after each response to return to input mode.

7 Using Programmatic Interfaces

Previous sections have described how to send textual commands to processes and retrieve textual results. This section describes how to access two facilities, the Subsystem Programmatic Interface (SPI) and the Event Management Service (EMS), that provide a programmatic interface for managing:

- ☐ Operating system utilities
- ☐ Tandem products such as Pathway, the TMF subsystem, and SNAX (also known as subsystems)
- ☐ Other applications that support these interfaces

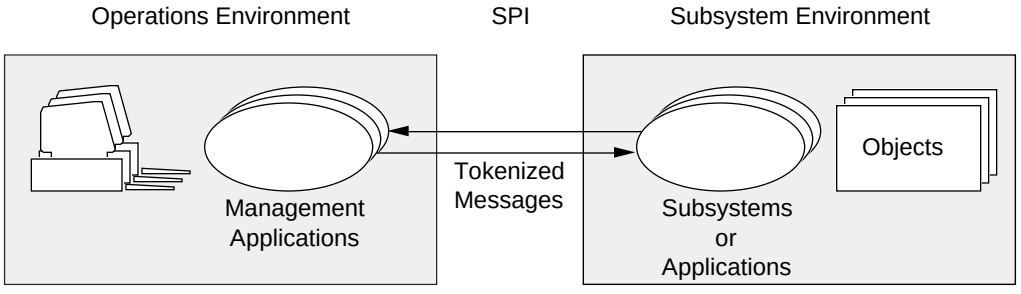
SPI and EMS are considered to be programmatic interfaces because information is represented by coded values, called tokens, which are easier for programs to manipulate than textual commands and results.

This section does not provide detailed information about SPI or EMS; it does, however, show how to use TACL to communicate with SPI and EMS. For additional information, see the *Introduction to Distributed Systems Management (DSM)*, the *Distributed Systems Management (DSM) Programming Manual*, and the *Event Management Service (EMS) Manual*.

Overview of SPI and EMS

SPI and EMS are two components of the Distributed System Management (DSM) product group, a set of software applications, tools, and services that facilitates management of systems and networks:

- ☐ SPI is a set of procedures, associated definition files, and programming conventions used for building, sending, retrieving, and decoding messages sent between management applications and the Tandem subsystems or business applications they manage. SPI makes possible the programmatic equivalent of an interactive command interface, allowing you to automate management tasks. The following diagram shows how applications interact with SPI:

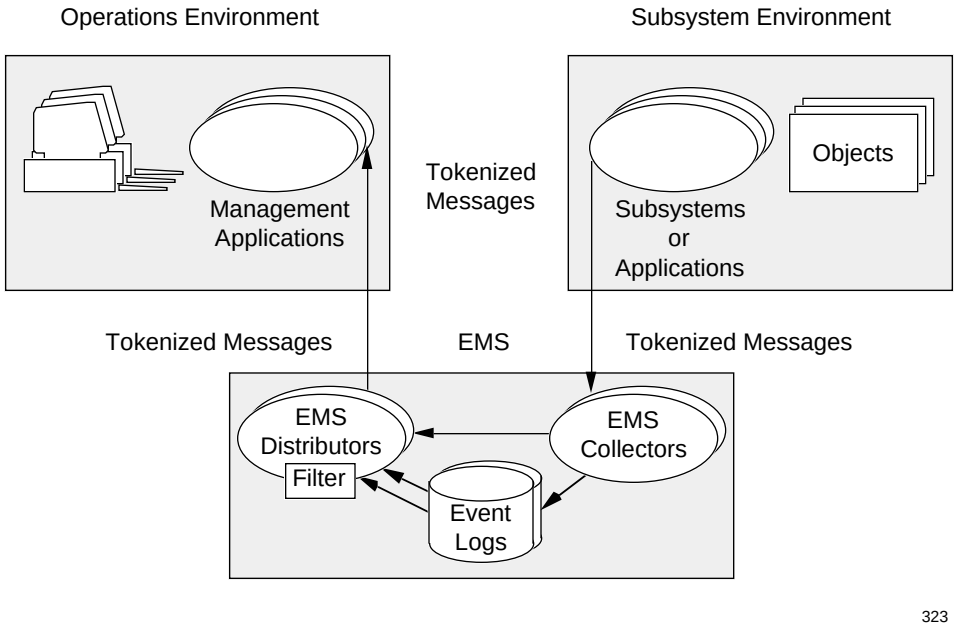


322

- ☐ EMS is a set of processes, associated definition files, tools, and programming conventions that provides event-message collection, logging, and distribution for the operating system. (An event is a significant change in some condition in the system or network, such as a device failure, a notification of limits exceeded, or a

request for action.) EMS allows you to monitor the status of system and application components. EMS messages are a type of SPI message; to communicate with EMS, you use SPI and EMS procedures.

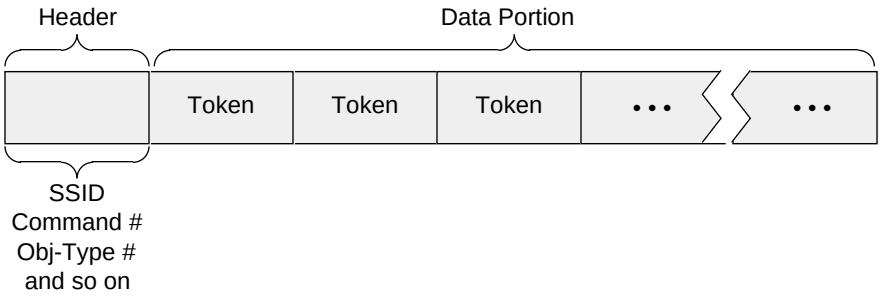
The following diagram shows a sample flow of EMS messages from subsystems to management applications. An application can also retrieve messages without involvement of distributor processes.



To transport messages between management applications and subsystems, you place tokens into a buffer. Each token consists of a token type and a token number—known collectively as a token code—and a token value. For example, a buffer that contains a Pathway PATHCOM START TCP command would contain the following tokens:

- Token code ZSPI-TKN-COMMAND, which signifies an SPI command, with a token value of ZPWY-CMD-START, indicating a request for a Pathway START command.
- Token code ZSPI-TKN-OBJECT-TYPE, which signifies an object type, with a token value of ZPWY-OBJ-TCP.
- Token code ZPWY-MAP-SEL-TCP, which signifies information about a TCP, with a token value of ZPWY-DDL-SEL-TCP. The program would assign the TCP name to the ZTCP field in the ZPWY-DDL-SEL-TCP structure.

The buffer transports the tokens to and from the target subsystem. The following is a simplified diagram of an SPI message. Again, each token contains a token code and a token value:



324 7

The ZSPI-TKN-COMMAND code and others listed previously are defined in files provided by Tandem (stored in ZSPIDEF and ZSPISEGF subvolumes). When you use SPI or EMS, you include definition files for the language you are using and subsystem-specific definition files for each subsystem you want to manage. To retrieve event messages, you also need the EMS definition file for the subsystem.

TACL provides built-in functions that initialize and manipulate the contents of message buffers. To establish communication with a subsystem, you use #REQUESTER or #SERVER with IN set to $RECEIVE to establish communication and #APPEND(V) and #EXTRACT(V) to send and receive a buffer, respectively.

SPI and EMS are based on the same message format, but the two interfaces differ in their purposes and uses; used together, they can provide a wide range of management services. For example, you can use EMS to monitor a large array of subsystems. If a significant situation arises, you can use SPI to initiate a dialog with the appropriate subsystem. You can also use TACL to learn about SPI and EMS interactively.

Using SPI

The Subsystem Programmatic Interface (SPI) allows you to store tokenized commands in buffers, send them to Tandem subsystems (such as Pathway), and receive a buffer containing tokenized replies. TACL uses standard SPI type names and supports standard SPI data types. Before sending or receiving an SPI message, you must:

- ☐ Know the subsystem ID for the desired subsystem.
- ☐ Know the names of individual tokens of interest.

You can find the necessary files in the ZSPIDEF subvolume supplied by Tandem; load them from within your TACL code or attach segment files (in the ZSPISEGF subvolume) that contain the information.

The functions in Table 7-1 provide access to SPI formats and protocols.

Table 7-1. TACL Functions That Support SPI

Function	Description
#SSINIT	Initializes a STRUCT as an SPI message buffer.
#SSGET	Retrieves token values from an SPI buffer, converts the values to external representation, and makes the results available.
#SSGETV	Retrieves token values from an SPI buffer, converts the values to external representation, and makes the results available. #SSGETV must be used for extensible structured tokens and tokens of type ZSPI^TYPE^STRUCT.
#SSPUT	Converts token values from external representation to binary form and puts them into an SPI buffer.
#SSPUTV	Converts token values in extensible structured tokens and tokens of type ZSPI^TYP^STRUCT.
#SSNULL	Initializes a structured token and sets the values to null.
#SSMOVE	Copies tokens from one message buffer to another by performing a sequence of #SSGETV and #SSPUTV operations.

To use SPI, follow these steps:

1. Define a message buffer (#DEF).
2. Initialize any structured tokens you plan to use (#SSNULL).
3. Initialize the buffer (#SSINIT).
4. Store requests in the buffer (#SSPUT, #SSPUTV).
5. Send the buffer (#APPENDV).
6. Retrieve results from the buffer (#SSGET, #SSGETV).

To copy tokens from one buffer to another, use #SSMOVE, which performs a sequence of #SSGETV and #SSPUTV operations.

TACL interacts with other processes as a server (through $RECEIVE) and as a requester (through the use of the #REQUESTER, #APPEND(V), and #EXTRACT(V) built-in functions). When using $RECEIVE, the IN and OUT files do not automatically provide an SPI interface, but can be made into an SPI interface by using an #INPUTV and #REPLYV loop protected by an exception handler.

Defining an SPI Buffer

SPI works with binary values. You declare an SPI buffer as a variable of type STRUCT. TACL allows you to refer to SPI values as text and translates them to and from binary as necessary.

Each Tandem subsystem includes an SPI definition file, named ZSPIDEF.*subsys*TACL. This definition file includes a buffer declaration named *subsys*^DDL^MSG^BUFFER. Allocate a buffer variable of at least the size of *subsys*^VAL^BUFLEN. Some subsystems use different buffer-size values for different commands; see the individual subsystem manuals for details. Using these definitions, your STRUCT definition might look like this:

```
[#DEF spi_buf STRUCT
  BEGIN
    BYTE buffer (1:ZTAC^VAL^BUFLEN);
  END;
]
```

When you use the definitions supplied by Tandem, your code retains compatibility with future changes in length.

TACL sends the length of the STRUCT to the SSINIT procedure and ignores the rest of the definition.

Note TACL has a maximum I/O buffer size of 5000 bytes and STRUCT variables are also limited to 5000 bytes. In addition, TACL cannot perform nowaited I/O to files opened by #REQUESTER for buffers larger than 239 bytes.

Subsystem ID

After you define a buffer, specify a subsystem ID—generally for the target subsystem—and a command. Include these items in a call to #SSINIT. The following statement initializes a buffer that contains a status command for the EMS subsystem:

```
#SET status [#SSINIT spi_buf [ZEMS^VAL^EXTERNAL^SSID]
  [ZEMS^CMD^STATUS] ]
```

The subsystem ID in TACL is specified as one to eight alphanumeric characters or hyphens that specify the subsystem owner, followed by a period separator, a subsystem number or a subsystem name, a period, and a version number. All three information fields are required. You must enter the subsystem owner exactly as specified for the subsystem; TACL does not case-shift it up or down. The first character must be alphabetic. TACL supports a special data type, ZSPI^TDT^SSID, for subsystem ID information.

TANDEM.EMS.0 and TANDEM.43.1245 are examples of valid subsystem IDs for Tandem subsystems.

The following considerations apply to the use of subsystem IDs in TACL:

- ☐ The null subsystem ID (all binary zeroes) is expressed as 0.0.0 in TACL.
- ☐ TACL performs automatic conversion between text and the internal format used by SPI for subsystem IDs.
- ☐ Your TACL function must specify the subsystem ID for each subsystem with which your application communicates.

Token Data Types

TACL supports the token data types listed in Table 7-2.

Table 7-2. SPI Token Data Types (Page 1 of 2)

Token Data Type Name	Value Type and Range
ZSPI^TDT^BOOL	-32768 – +32767
ZSPI^TDT^BYTE	0 – 255
ZSPI^TDT^CHAR	Characters
ZSPI^TDT^CRTPID	Any valid process name or *cpu*,*pin*. Systems whose numbers cannot be found are given number 255 on input; system number 255 is shown as \?? on output. The creation time of any unnamed process or the *cpu*,*pin* of a named process can be accessed only by redefining the fields appropriately.
ZSPI^TDT^DEVICE	Any valid device name. Systems whose numbers cannot be found are given number 255 on input; system number 255 is shown as \?? on output.
ZSPI^TDT^ENDLIST	Not applicable.
ZSPI^TDT^ENUM	-32768 to +32767
ZSPI^TDT^ERROR	A subsystem ID, a period, and a number in the range -32768 to +32767.
ZSPI^TDT^FNAME	Any valid file name; missing fields are not defaulted. Systems whose numbers cannot be found are given number 255 on input; system number 255 is shown as \?? on output.
ZSPI^TDT^FNAME32	A 32-bit file name of the form used by Distributed Name Service.
ZSPI^TDT^INT	-32768 to +32767
ZSPI^TDT^INT2	-2147483648 to +2147483647
ZSPI^TDT^INT4	-(2**63) to +(2**63)-1
ZSPI^TDT^LIST	Not applicable
ZSPI^TDT^MAP	Must be put into, or obtained from, a STRUCT. Definition of the STRUCT provides rules for conversion between internal and external formats.

Table 7-2. SPI Token Data Types (Page 2 of 2)

Token Data Type Name	Value Type and Range
ZSPI^TDT^SSCTL	0 to 255
ZSPI^TDT^SSID	Subsystem owner, period, subsystem number, period, and version number.
ZSPI^TDT^STRUCT	Must be put into, or obtained from, a STRUCT. Definition of the STRUCT provides rules for conversion between internal and external formats.
ZSPI^TDT^SUBVOL	Any valid subvolume name; missing fields are not defaulted. Systems whose numbers cannot be found are given number 255 on input; system number 255 is shown as \?? on output.
ZSPI^TDT^TIMESTAMP	-(2**63) to +(2**63)-1
ZSPI^TDT^TOKENCODE	-2147483648 to +2147483647. On input, a 32-bit STRUCT can be used.
ZSPI^TDT^TRANSID	A TMF transaction identifier formatted as follows: *system-name*(*crash-count*).*cpu*.*sequence* If the internal-format *trans-id* contains a crash count of zero, the transaction ID is formatted as follows: *system-name*.*cpu*.*sequence* In either case, if the system is not named, the transaction ID is formatted as follows: *cpu*.*sequence* If the name of a system cannot be found on output, *system-name* is replaced by *system-number*.
ZSPI^TDT^UINT	0 to 65535
ZSPI^TDT^USERNAME	Any valid user name; a user number cannot be used.

TACL interprets an unknown token data type as if the data type were ZSPI^TDT^INT.

Variable-Length Data Items

In TACL, the byte length appears before the data, separated from it by a space.

Token Codes

In TACL, a token code is an integer, a numeric variable, or a two-word STRUCT. Token code definitions, as generated by DDL, are included in the TACL versions of the SPI and subsystem definition files. You can refer to the individual fields of a token code by using a STRUCT that is defined as follows:

```
[#DEF token STRUCT
  BEGIN
  INT2    code;
  STRUCT fields REDEFINES int32;
    BEGIN
    BYTE    datatype;
    BYTE    bytelength;
    INT     number;
    END;
  END;
]
```

Token codes cannot be composed or decomposed by simple arithmetic, because the token number is signed and simple arithmetic would extend the sign.

Token Maps

In TACL, a token map is stored in a STRUCT variable. The definition of the STRUCT is irrelevant to TACL.

Token map definitions, as generated by DDL, are included in the TACL definition files for subsystems that define extensible structured tokens.

When using a token map, TACL verifies that the contents of the token map are consistent with the size of the STRUCT in which it is stored.

External Representation of Token Values

Use text (instead of binary bits) when storing and retrieving token values. Legal text characters come from the ISO 8859.1 character set. The following lists conditions of data representation:

- If the token-length of a token is 255, the token value is of variable length. TACL represents such a token value as a number indicating the one-word byte length of the token, followed by a space, followed by the token value or values.
- If the data type of a token is ZSPI^TDT^CHAR, its token value is represented by a number of contiguous characters; that number is equal to the actual length of the token. (This is the value of the token-length field if that value is less than 255, or the actual length in bytes if the token-length field is 255.)
- If the data type of a token is ZSPI^TDT^STRUCT or ZSPI^TDT^MAP, you must handle the token with #SSGETV and #SSPUTV so that the binary data can be moved to and from STRUCT variables whose definitions provide the appropriate conversions to and from external representations.

If the data type of a token is any other than those already listed, the token value is represented by a space-separated list *m/n* items, where *m* is the actual length of the token and *n* is the basic length of the token data type. If the actual token length is not evenly divisible by the basic data-type length, the last bytes cannot be set or seen. TACL shows each item in its usual external representation.

All elements pertaining to the Subsystem Programmatic Interface are described in detail in the *Distributed Systems Management (DSM) Programming Manual.*

Using SPI Functions

The following definition, when used with #EXTRACT(V), allows you to extract fields from a subsystem ID:

```
?SECTION template STRUCT
BEGIN
 SSID  ss;
 STRUCT   z_ssid REDEFINES ss;
       BEGIN
          CHAR  z_owner (1:8);
          INT   z_number;
          UINT  z_version;
       END;
END;
#DEF ss1 STRUCT LIKE template;
```

Use the code in Figure 7-1, `same_ssid`, to compare two subsystem IDs returned by #SSGETV with ZSPI^TKN^NEXTCODE or ZSPI^TKN^NEXTTOKEN. The code uses the `template` STRUCT defined under “Defining an SPI Buffer,” earlier in this section, and ignores the version field. To invoke this routine, load the file and type:

```
SAME_SSID ssid1 ssid2
```

Figure 7-1. Comparing Two Subsystem IDs

```
?SECTION same_ssid ROUTINE
#FRAME
#PUSH sstext
#DEF ss1 STRUCT LIKE template;
#DEF ss2 STRUCT LIKE template;
#IF {SINK} [#ARGUMENT / VALUE sstext/ SUBSYSTEM]
#SET ss1 [sstext]
#IF {SINK} [#ARGUMENT / VALUE sstext/ SUBSYSTEM]
#SET ss2 [sstext]
#RESULT [#COMPUTE  [#COMPAREV  ss1:z_ssid:z_owner(1:8)
                               ss2:z_ssid:z_owner(1:8)]
          AND      [#COMPAREV  ss1:z_ssid:z_number
                               ss2:z_ssid:z_number] ]
#UNFRAME
```

Use the routine in Figure 7-2, dumplog, to retrieve the name of the current EMS log file using SPI. The example then calls EMSDIST to read the current log and display the message on the home terminal. To invoke this routine, load the associated file and enter:

```
dumplog [time]
```

Figure 7-2. Displaying the EMS Log (Page 1 of 2)

```
?SECTION dumplog ROUTINE
#FRAME
[#PUSH
  err                          == Error return value
  io_err                       == I/O Error return value
  reply                        == Reply I/O variable
  request                      == Request I/O variable
  retcode                      == Return code from server
  ZEMS^CMD^STATUS              == ZEMS status command
  ZEMS^VAL^EXTERNAL^SSID       == ZEMS subsystem ID
  zspidef                      == zspidef subvolume
  arg
  rslt
  defs
]
== Include SPI and EMS definitions
============================================
#LOAD /LOADED defs, KEEP 1/ $system.zspidef.zemstacl
#LOAD /LOADED defs, KEEP 1/ $system.zspidef.zspitacl
#LOAD /LOADED defs, KEEP 1/ $system.zspidef.ztactacl

== Define the SPI buffer
[#DEF spi_buf STRUCT
  BEGIN
    BYTE b(1:1024);
  END;
]
== Initialize the SPI buffer; EMS subsystem, STATUS command
#SET err [#SSINIT spi_buf [ZEMS^VAL^EXTERNAL^SSID]
  [ZEMS^CMD^STATUS] ]
[#IF err |THEN|
  #OUTPUT *** Error [err] from #SSINIT]

== Retrieve the name of the EMS log file
#SET err [#REQUESTER /WAIT/
                      READ $0.#ZSPI io_err reply request]
[#IF err |THEN|
  #OUTPUT *** Error [err] opening $0.#ZSPI
  #UNFRAME
  #RETURN
]
```

Figure 7-2. Displaying the EMS Log (Page 2 of 2)

```
== Send the request (the open was waited, so we do not need
== to wait for the variable to become empty.)
#APPENDV request spi_buf

== Retrieve the results
#EXTRACTV reply spi_buf
[#IF NOT [#EMPTYV io_err] |THEN|
  #OUTPUT *** Error [io_err] sending to $0.#ZSPI
  #UNFRAME
  #RETURN
]

== Close SPI
#SET err [#REQUESTER CLOSE request]
[#IF err |THEN|
  #OUTPUT *** Error [err] closing $0.#ZSPI
  #UNFRAME
  #RETURN
]

== Retrieve the return code and place it into the SPI buffer
#SETMANY err _ retcode , [#SSGET /INDEX 1/ spi_buf
  ZSPI^TKN^RETCODE]
[#IF err |THEN|
  #OUTPUT *** Error [err] from #SSGET ZSPI^TKN^RETCODE
  #UNFRAME
  #RETURN
]
== Retrieve EMS status information and place it into the SPI
== buffer
#SETMANY err _ , [#SSGETV /INDEX 1/ spi_buf
                  ZEMS^MAP^COL^STATUS
                  ZEMS^DDL^COL^STATUS]
[#IF err |THEN|
  #OUTPUT *** Error [err] from #SSGETV ZEMS^MAP^COL^STATUS
  #UNFRAME
  #RETURN
|ELSE|
  == Retrieve the time argument (without checking contents)
  #SET rslt [#ARGUMENT /TEXT arg/ WORD END]
    == Call EMSDIST
    emsdist logfile &
    [ZEMS^DDL^COL^STATUS:ZCOL^CURRENTFILENAME], &
      [#IF NOT [#EMPTY [arg]] |THEN| time [arg], ] &
      type printing, textout [#MYTERM]
]
#UNFRAME
```

The following session shows sample output:

```
17>dumplog 16:50
92-05-28 16:51:22 \SYS.003,000 TANDEM.MSGSYS.C20 000104
                       Receive Queue for CPU 3,
                       PIN 50 Greater Than 10 LCBs
92-05-28 17:00:34 \SYS.004,013 TANDEM.EMS.C20 000165 LDEV
                       0056 CU %120 CSS SUBDEVICE ERROR,
                       CIU SUBDEV #000003 F, M=%043200
                       STATUS %000005
```

To generate a request for EMS, see the following subsection, "Using EMS."

Using EMS

EMS provides a centralized event collection, logging, and distribution facility for Tandem networks. Subsystems within the NonStop kernel environment can define and generate tokenized event messages. For additional information about EMS, see the *Event Management Service (EMS) Manual.*

Communicating With EMS

The built-in functions in Table 7-3 provide access to EMS.

Table 7-3. Functions That Support EMS

Function	Description
#EMSINIT(V)	Builds the header and initializes a STRUCT as an event message buffer, so that you can use the STRUCT in calls to other #EMSxxx and #SSxxx functions.
#EMSGET(V)	Retrieves token values from a buffer, converts them to external representation, and returns the external representation as its result. This function is similar to #SSGET.
#SSGET(V)	Retrieves token values from an SPI buffer, converts them to external representation, and makes the results available. Use #SSGETV (instead of #SSGET) for extensible structured tokens and tokens of type ZSPI^TYPE^STRUCT.
#SSPUT	Converts token values from external representation to binary form and puts them into an SPI buffer.
#SSPUTV	Converts token values in extensible structured tokens and tokens of type ZSPI^TYP^STRUCT.
#SSNULL	Initializes a STRUCT and sets the values to null.
#SSMOVE	Copies tokens from one message buffer to another by performing a sequence of #SSGETV and #SSPUTV operations.
#EMSTEXT(V)	Obtains information from an event buffer and makes it available in printable text form.
#EMSADDSUBJECT(V)	Adds a subject token to an event message buffer.

Before communicating with EMS, you must:

- ☐ Know the subsystem ID for EMS
- ☐ Know the names of individual tokens of interest

You can find the necessary definitions in the ZSPIDEF subvolume; load them from within your TACL code or create and attach segment files (in the ZEMSSEGF subvolume) that contain the information.

Follow these steps to generate an EMS event:

1. Define a message buffer (#DEF)
2. Initialize the buffer (#EMSINIT)
3. Store requests in the buffer and send the buffer (#SSPUT, #SSPUTV)

Generating an EMS Event

Use the macro in Figure 7-3, `taclevent`, to generate an event from EMS and, optionally, set the action flag or critical flag. The syntax is:.

```
taclevent urgency
```

where *urgency* is 1 for no flag, 2 for an action flag, and 3 for a critical flag.

You can use the `dumplog` routine (n Figure 7-2) to display the event log and check your results.

Figure 7-3. Generating an EMS Event (Page 1 of 3)

```
?SECTION taclevent MACRO
#FRAME

#PUSH evt_x evt_error req_error req_read req_prompt
#PUSH action_id collector

== Load the files.  (If the files do not exist, TACL
== returns an error and exits.)
#LOAD /LOADED evt_x/ $system.zspidef.zemstacl
#LOAD /LOADED evt_x/ $system.zspidef.zspitacl
#LOAD /LOADED evt_x/ $system.zspidef.ztactacl

[#DEF evt_buf STRUCT
  BEGIN
    BYTE b(1:3000);
  END;
]

[#DEF build_evt ROUTINE |BODY|
  #FRAME
  #PUSH evt_num action emphasis evt_text text_len

  SINK [#ARGUMENT /TEXT evt_num/   NUMBER]
  SINK [#ARGUMENT /TEXT action/    NUMBER]
  SINK [#ARGUMENT /TEXT emphasis/  NUMBER]
  SINK [#ARGUMENT /TEXT evt_text/  TEXT  ]
```

Figure 7-3. Generating an EMS Event (Page 2 of 3)

```
== Build the event and initialize the EMS buffer
[#SET evt_error [#EMSINIT evt_buf [ZEMS^VAL^EXTERNAL^SSID]
      [evt_num] ZEMS^TKN^TEXT [#charcount evt_text]
      [evt_text]]
]
[#IF [evt_error] |THEN|
  #OUTPUT *ERROR* EMSINIT error [evt_error]
  #RETURN
]

== If emphasis is TRUE, set the CRITICAL flag
[#IF [emphasis] |THEN|
  #SET evt_error [#SSPUT evt_buf ZEMS^TKN^EMPHASIS &
    [ZSPI^VAL^TRUE]]
  [#IF [evt_error] |THEN|
    #OUTPUT *ERROR* SSPUT error [evt_error] on &
      Emphasis token
    #RETURN
  ] == End IF
] == End IF

== If action is TRUE, set the ACTION-NEEDED flag
[#IF [action] |THEN|
  #SET evt_error [#SSPUT evt_buf ZEMS^TKN^ACTION^NEEDED &
    [ZSPI^VAL^TRUE]]
  [#IF [evt_error]  |THEN|
    #OUTPUT *ERROR* SSPUT error [evt_error] on &
      Action-Needed token
  ] == End IF

  == SET the action-ID token to a unique value
  #SET evt_error [#SSPUT evt_buf ZEMS^TKN^ACTION^ID
    [action_id]]
  [#IF [evt_error] |THEN|
    #OUTPUT *ERROR* SSPUT error [evt_error] on Action-ID
    #RETURN
  |ELSE| #SET action_id [#COMPUTE action_id + 1]
  ]
]

== Send the event buffer to the Collector
#APPENDV req_prompt evt_buf
#UNFRAME
] == End of build_evt
```

Figure 7-3. Generating an EMS Event (Page 3 of 3)

```
== Main logic
================================================
#SET collector $0
#SET action_id 0

== Open the Collector for WRITEREADs; $0 expects a WRITEREAD
#SET evt_error [#REQUESTER /WAIT/ READ [collector] &
                req_error req_read req_prompt]

== If open fails, display an error message.  Otherwise send
== an event message.
[#IF [evt_error] |THEN|
  #OUTPUT *ERROR* #REQUESTER OPEN error [evt_error]
| ELSE |
  [#CASE %1%
    |1|
      build_evt 9997  0  0  Test: Calm event
    |2|
      build_evt 9998 -1  0  Test: Action event
    |3|
      build_evt 9997  0 -1  Test: Critical event
    |OTHERWISE|
      #OUTPUT Invalid argument.  Must be:
      #OUTPUT 1 (event),
      #OUTPUT 2 (action event), or
      #OUTPUT 3 (critical event).
  ]
]

== Close the Collector
#SET evt_error [#REQUESTER /WAIT/ close req_read]
[#IF [evt_error] |THEN|
  #OUTPUT *ERROR* #REQUESTER CLOSE error [evt_error]
]
#UNFRAME
```

(This page left intentionally blank)

8 Example of a System Management Program

The use of TACL for system management purposes combines several tasks:

- ☐ Starting, stopping, and monitoring processes
- ☐ Communicating with processes
- ☐ Acting on responses
- ☐ Generating commands
- ☐ Handling errors and other exceptions

This section contains an example of a system management program that checks the status of several system elements.

Monitoring System Operation

The macro in Figure 8-1, ckup, illustrates one way to check the status of the following system elements:

- ☐ CPUs
- ☐ Disk space
- ☐ The spooler
- ☐ TMF
- ☐ Device problems: bad sectors and other device errors

The macro, ckup, uses #CHARxxx routines to retrieve information. To run this macro, load the associated file and enter:

```
ckup
```

Portions of ckup require a version of TACL released at C20 or later. In addition, your TACL process must be named.

The macro runs as a batch file; you start it and it runs through several tests, displaying results. Rather than placing defines at the start of the program, the program defines procedural variables near the code that calls them.

Other ways to structure system management programs include:

- ☐ A menu interface, providing selections for various subsystems.
- ☐ An interactive interface that asks questions and performs more detailed checking, depending on your responses.
- ☐ The use of SPI and EMS facilities, as described in Section 7, “Using Programmatic Interfaces.”

Figure 8-1. Monitoring System Status (Page 1 of 12)

```
?SECTION ckup MACRO
#FRAME

== Definitions for terminal I/O

[#DEF esc STRUCT
  BEGIN
    BYTE byte_escape VALUE 27;
    CHAR char_escape REDEFINES byte_escape;
  END;
] == end def

== The following #DEFs must be on single lines for
== later replacement in #OUTPUT calls
#PUSH rev end termout
#SET rev [Esc:Char_Escape]6$    == Terminal Control reverse
#SET end [Esc:Char_Escape]6`    == Terminal Control end

[#DEF Tcr text |BODY| [#IF [termout] |THEN|[rev]] ]

[#DEF Tce text |BODY| [#IF [termout] |THEN|[end]] ]

#PUSH #PMSG  == turn off #PMSG flag
#SET  #PMSG 0

[#IF NOT [#EMPTY %1%] |THEN|
  #PUSH #OUT
  #SET termout 0
  [#IF [#MATCH p %1%] |THEN|
    #SET #OUT $s.#hum
  |ELSE|
    #SET #OUT %1%
  ] == end if
|ELSE|
  #SET termout -1
] == end if
```

Note If the result of an expansion must fit on one line, the related definition must fit onto one line. The definitions of `tcr` and `tce` in Figure 8-1 follow this rule.

Figure 8-1. Monitoring System Status (Page 2 of 12)

```
== ---------------------------------------------------------
== Display general system information
== ---------------------------------------------------------

#PUSH release version
#OUTPUT
#OUTPUT status for [#MYSYSTEM]
#OUTPUT [_CONTIME_TO_TEXT_DATE [#CONTIME [#TIMESTAMP]]]
#SETMANY release version , [#TOSVERSION]
[#IF [#MATCH [release] L ] |THEN| #SET release B ]
[#IF [#MATCH [release] M ] |THEN| #SET release C ]
#OUTPUT OPSYS: [release][version]
#OUTPUT SYSnn: &
 [#FILEINFO/SUBVOL/[#PROCESSINFO/PROGRAMFILE/0,0]]
#POP release
#OUTPUT

== ---------------------------------------------------------
==  Display whether configured processors
==  are up or down, generalized to run on all systems.
== ---------------------------------------------------------

[#DEF cpuupordown ROUTINE |BODY|
  #FRAME
  #PUSH cpucounter cpustatus maxcpus type
  #SET cpucounter 0
   SINK [#ARGUMENT /TEXT maxcpus/ NUMBER] == cpu count
  [#LOOP |WHILE| cpucounter < maxcpus |DO|
    SINK [#ARGUMENT /TEXT cpustatus/ NUMBER]
    [#CASE [#PROCESSORTYPE [cpucounter]]
      | -2  | #SET type Unknown type
      | -1  | #SET type Nonexistent
      |  0  | #SET type TNS1
      |  1  | #SET type TNS2
      |  2  | #SET type TXP
      |  3  | #SET type VLX
      |  4  | #SET type CLX
      |  5  | #SET type CYCLONE
    ]  == end of CASE
```

Figure 8-1. Monitoring System Status (Page 3 of 12)

```
    == For CPU status, -1 = up and 0 = down
    [#IF cpustatus |THEN|
      #OUTPUT [type] Processor [cpucounter] is up.
    |ELSE|
      #OUTPUT [Tcr][type] Processor [cpucounter] &
is down.[Tce]
    ] == end if
    #SET cpucounter [#COMPUTE cpucounter + 1]
  ] == end loop
#UNFRAME
] == end define

cpuupordown [#PROCESSORSTATUS]
#OUTPUT

== ------------------------------------------------------
== Display the file size of OPRLOG and ESLOG
== ------------------------------------------------------

#PUSH filename maxext size termout eof pri sec

#SET filename $SYSTEM.SYSTEM.OPRLOG
[#IF [#FILEINFO/EXISTENCE/[filename]] |THEN|
  #SETMANY eof pri sec, [#FILEINFO/EOF,PRIMARY,SECONDARY/&
    [filename]]
  #SET maxext [#COMPUTE [#FILEINFO/MAXEXTENTS/[filename]] &
    - 1]
  #SET size [#COMPUTE ( eof*100 ) / (( pri + &
    (maxext*sec )) * 2048 )]
  [#IF termout AND (size = 100) |THEN|
    #OUTPUT [rev]OPRLOG is [size] percent full.[end]
  |ELSE|
    #OUTPUT OPRLOG is [size] percent full.
  ] == end if
|ELSE| == oprlog does not exist.
] == end if

#SET filename $SYSTEM.SYSTEM.ESLOG
[#IF [#FILEINFO/EXISTENCE/[filename]] |THEN|
  #SETMANY eof pri sec, [#FILEINFO/EOF,PRIMARY,SECONDARY/&
    [filename]]
  #SET maxext [#COMPUTE [#FILEINFO/MAXEXTENTS/[filename]] &
    - 1]
  #SET size [#COMPUTE ( eof*100 ) / (( pri + &
    (maxext*sec )) * 2048 )]
  #OUTPUT ESLOG is [size] percent full.
|ELSE| == eslog does not exist.
] == end if
```

Figure 8-1. Monitoring System Status (Page 4 of 12)

```
#POP filename maxext size termout eof pri sec
#OUTPUT

== ------------------------------------------------------
== Display an analysis of disk space
== ------------------------------------------------------

#PUSH maxfragments minfreespace addr dsapout dsapout2
#PUSH disk pfree line fragments

== Set the number to flag for maximum fragments:
#SET maxfragments 100
== Set the percent minimum free space to flag
#SET minfreespace 10

#OUTPUT Disk space analysis:
#OUTPUT /HOLD/ disks > [#COMPUTE 100 - [minfreespace]] &
 percent full
#OUTPUT ~_and > [maxfragments] fragments ...
DSAP /OUTV dsapout/ *,SHORT

== Remove banner lines from output
#EXTRACTV dsapout line
[#LOOP |WHILE| NOT [#MATCH volume* [line]] |DO|
  #EXTRACTV dsapout line
] == end loop

== Remove lines with "unavailable" disks
#SET addr [#CHARFIND dsapout 1 unavailable]
[#LOOP |WHILE| [addr]  |DO|
  #LINEDEL dsapout [#LINEADDR dsapout [addr]]
  #SET addr [#CHARFIND dsapout [addr] unavailable]
] == end loop

== Copy for fragment count
COPYVAR dsapout dsapout2
[#LOOP |WHILE| NOT [#EMPTYV dsapout] |DO|
  #EXTRACTV dsapout line                    == data line
  #SET disk [#CHARGET line 1 FOR 8]         == disk name
  #SET pfree [#CHARGET line 38 FOR 2] == percent
  [#IF [pfree] < [minfreespace] |THEN|
    #OUTPUT /HOLD/[Tcr][disk]
    #OUTPUT /COLUMN 13/ is [#COMPUTE 100 - [pfree]] &
percent full.[Tce]
  ] == end if
]  == end loop
```

Figure 8-1. Monitoring System Status (Page 5 of 12)

```
[#LOOP |WHILE| NOT [#EMPTYV dsapout2] |DO|
  #EXTRACTV dsapout2 line             == data line
  #SET disk [#CHARGET line 1 FOR 8]   == disk name
  #SET fragments [#CHARGET line 45 FOR 4] == fragment count
  [#IF fragments > [maxfragments] |THEN|
#OUTPUT/hold/[Tcr][disk]
#OUTPUT /COLUMN 13/ has [fragments] fragments.[Tce]
  ] == end if
] == end loop

#POP maxfragments minfreespace addr dsapout dsapout2
#POP disk pfree line fragments
#OUTPUT

== ------------------------------------------------------------
== Check specified files for index level growth and
== for file > 90% full
== ------------------------------------------------------------

[#DEF lvl MACRO |BODY|
  #FRAME
  == Check for index level increase growth
  == Syntax:   LVL filename acceptable_index_level_number
  == Example:  LVL DATABASE 3

  #PUSH stats line level eof maxbytes percent
  #OUTPUT Checking %1%
  SQLCI/OUTV stats/FILEINFO %1%,STAT;
  [#LOOP |WHILE| NOT [#MATCH LEVEL* [line] OR
      [#EMPTY [stats]]] |DO|
    #EXTRACTV stats line
  ] == end loop
  #EXTRACTV stats line
  #SETMANY level, [line]
  [#IF NOT [#MATCH ERROR [level]] |THEN|
    [#IF [level] > %2% |THEN|
      #OUTPUT %1% has grown from %2% index levels to &
[level].
    ]
  ]

  #SETMANY eof maxbytes, [#FILEINFO /EOF,MAXBYTES/ %1%]
  #SET percent [#COMPUTE [eof]*100/[maxbytes]]
  [#IF [percent] > 90 |THEN|
    #OUTPUT %1% is [percent] percent full.
  ] == end if
#UNFRAME
```

Figure 8-1. Monitoring System Status (Page 6 of 12)

```
] == end def
#OUTPUT

== ------------------------------------------------------
== Report on databases.
== (Insert database files that you want to monitor.)
== The format is:  filename AcceptableIndexlevelNumber
== Example:  LVL $SYSTEM.SYSTEM.USERID 1
== ------------------------------------------------------

== ------------------------------------------------------
== Display Spoolcom information
== ------------------------------------------------------
#PUSH deverror spoolout line collector state size device
#PUSH errornum
#SET deverror 0
#OUTPUT
#OUTPUT SPOOLCOM information:
SPOOLCOM /OUTV spoolout/ ; COLLECT ; DEV
#EXTRACTV spoolout line == banner
#EXTRACTV spoolout line == blank line
#EXTRACTV spoolout line == first collector
[#LOOP |WHILE| NOT [#EMPTY [line]] |DO|
  #SETMANY collector state, [line]
  #SET size [#CHARGET line 74 FOR 2]
  [#IF (size > 90) |THEN|
    #OUTPUT /hold/ [Tcr]collector [collector] is [state]
    #OUTPUT ~_AND is [size] percent full.[Tce]
  ] == end if
  #EXTRACTV spoolout line == get next collector
] == end loop

#OUTPUT Checking devices ...
#EXTRACTV spoolout line == banner with device state etc.
#EXTRACTV spoolout line == first device
[#LOOP |WHILE| NOT [#EMPTYV line] |DO|
  #SETMANY device state errornum, [#CHARGET line 1 FOR &
    50]
  [#IF NOT [#MATCH WAITING [state]] AND NOT [#MATCH JOB* &
  [state]] |THEN|
    #OUTPUT /HOLD/ [Tcr]device [device]
    #OUTPUT /COLUMN 25/ is [state] [errornum][Tce]
    #SET deverror -1
  ] == end if
  #EXTRACTV spoolout line == get next device
] == end loop
```

Figure 8-1. Monitoring System Status (Page 7 of 12)

```
[#IF NOT deverror |THEN|
  #OUTPUT
  #OUTPUT All Spooler devices OK.
] == end if
#POP deverror spoolout line collector state size device
#POP errornum
#OUTPUT

== ------------------------------------------------------
==Display TMF information
== ------------------------------------------------------
#PUSH volok stat line volume vol status file code
#PUSH eof primary secondary code full
#SET volok -1

#OUTPUT TMF information:
#OUTPUT Collecting TMF status...
TMFCOM /OUTV stat/ STATUS TMF; STATUS VOLUMES
#EXTRACTV stat line == get first status line
[#LOOP |WHILE| NOT [#MATCH volume [line]] |DO|
  #OUTPUTV line
  #EXTRACTV stat line
] == end loop
#EXTRACTV stat line == underscore
#EXTRACTV stat line == first valid status volume line
[#LOOP |WHILE| NOT [#EMPTYV stat] |DO|
  #SETMANY vol _ status, [line]
  [#IF [#MATCH DISABLED [status]] |THEN|
    #OUTPUT
    #OUTPUT [vol] is [status]
    #SET volok 0
  ] == end if
  #EXTRACTV stat line
] == end if
[#IF volok |THEN|
  #OUTPUT
  #OUTPUT All TMF volumes OK.
] == end if
```

Figure 8-1. Monitoring System Status (Page 8 of 12)

```
== Get the percent full for TMF audit trail files
#OUTPUT
[#LOOP |DO|
   #SET file [#FILENAMES/MAXIMUM 1, PREVIOUS
[file]/$*.AUDIT.*]
   [#IF (NOT [#EMPTYV file]) |THEN|
     [#SETMANY eof primary secondary code ,
       [#FILEINFO/EOF,PRIMARY,SECONDARY,CODE /[file]]
     ] == end setmany
     [#IF ([code] = 134) |THEN|
       #SET full [#COMPUTE (eof*100) / &
         ((primary +(15*secondary)) * 2048)]
       [#IF [termout] AND ([full] = 100) |THEN|
         #OUTPUT [rev][file] is 100 percent full.[end]
       |ELSE|
         #OUTPUT [file] is [full] percent full.
       ] == end if termout
     ] == end if code
   ] == end if not emptyv
|UNTIL| ([#EMPTYV file])
] == end loop
#POP volok stat line volume vol status file code
#POP eof primary secondary code full
#OUTPUT

== ------------------------------------------------------
== Display disk status.
== ------------------------------------------------------
== Use PUP to get the status.
== Save output in pup_out.
== Loop, ignoring blank lines, and tokenize each line.
== Display paths in the "H", "D", "R", and "S" states.
== Display paths that are not "P" or "M",the preferred paths.

== Due to the format change of PUP output, this portion
== will work for C20 or later versions of PUP only.

#PUSH scanline badsectorflag puplistdev pupcommand
[#IF [version] >= 20 |THEN|
  #OUTPUT PUP information:
  #SET badsectorflag 0
  PUP /OUTV puplistdev/ LISTDEV DISC == collect data
  #EXTRACTV puplistdev scanline     == remove banner
  #EXTRACTV puplistdev scanline     == copy right
  [#LOOP |WHILE| NOT [#EMPTYV puplistdev] | DO |
    #EXTRACTV puplistdev scanline == get one line
```

Figure 8-1. Monitoring System Status (Page 9 of 12)

```
      [#IF NOT [#CHARFINDV scanline 6 "?"] |THEN|
        [#IF NOT [#CHARFINDV scanline 8 "-P"]  AND
             NOT [#CHARFINDV scanline 8 "-M "] AND
                 [#CHARFINDV scanline 18 " *"]
        |THEN| #OUTPUT [#CHARGET scanline 6 FOR 10] using &
backup path.
        ] == end if
        [#CASE [#CHARGET scanline 18 FOR 1]
          |D| #OUTPUT [Tcr][#CHARGET scanline 6 FOR 10] &
 disk down[Tce]
          |H| #OUTPUT [Tcr][#CHARGET scanline 6 FOR 10] &
 Hard down[Tce]
          |R| #OUTPUT [Tcr][#CHARGET scanline 6 FOR 10] &
 reviving[Tce]
          |S| #OUTPUT [Tcr][#CHARGET scanline 6 FOR 10] &
 Special state[Tce]
          |I| #OUTPUT [Tcr][#CHARGET scanline 6 FOR 10] &
 Inaccessible[Tce]
          |OTHERWISE| [#IF ([#CHARFINDV scanline 8 "-P"] OR
            [#CHARFINDV scanline 8 "-M "]) == accept -p or -m
            |THEN|
              == send a pup listbad command
              #APPEND pupcommand listbad [#CHARGET scanline 6
                FOR 10]
            ] == end if
          ] == end case
     ] == end if charfind
] == end loop
#POP version scanline badsectorflag puplistdev
#OUTPUT

== ------------------------------------------------------
== Check for bad disk sectors
== ------------------------------------------------------
#PUSH pupout addr badsectorflag linecounter
#PUSH maxlines

[#DEF msg STRUCT
  BEGIN
  CHAR disk(0:12);
  CHAR ok(0:1) VALUE OK;
  END;
] == end def

#OUTPUT Checking for bad sectors on all disks ...
PUP /INV pupcommand, OUTV pupout/
```

Figure 8-1. Monitoring System Status (Page 10 of 12)

```
#LINEDEL pupout 1 for 2 == delete banner

== Remove "#listbad"
#SET addr [#CHARFIND pupout 1 #LISTBAD]
[#LOOP |WHILE| [addr] > 0 |DO|
  #CHARDEL pupout [addr] for 9
  #SET addr [#CHARFIND pupout addr #listbad]
] == end loop

#SETMANY badsectorflag linecounter, 0 1
#SET maxlines [#LINECOUNT pupout]
[#IF [#MATCH #EXIT [#LINEGET pupout [maxlines] for 1] ]
 |THEN|
   #LINEDEL pupout [maxlines] == delete last line (#exit)
    #SET maxlines [#LINECOUNT pupout]
] == end if
#SET msg:disk(0:12) [#LINEGET pupout 1 for 1]
[#LOOP |WHILE| linecounter <= maxlines
  | DO  |
  [#IF [#MATCH NO* [#LINEGET pupout [linecounter] for 1] ]
  |THEN|
    #LINEDEL pupout [linecounter] ==remove "NO BAD SECTORS"
    #SET linecounter [#COMPUTE linecounter -1]
    #SET maxlines [#COMPUTE maxlines - 1]
    #LINEDEL pupout linecounter   == remove line $volume-x
    #LINEINS pupout [linecounter] [msg] == insert structure
  |ELSE|
    == Prepare structure for next possible OK disk
    [#IF [#MATCH $* [#LINEGET pupout [linecounter] for 1]]
    |THEN|
      #SET msg:disk(0:12) [#LINEGET pupout [linecounter]
        for 1]
    |ELSE|
      #SET badsectorflag -1 == flag to output entire report
    ]
  ] == end if, end if
  #SET linecounter [#COMPUTE linecounter + 1]
] == end loop

[#IF [badsectorflag]
  |THEN| #OUTPUTV pupout
  |ELSE| #OUTPUT
         #OUTPUT All disk sectors OK.
] == end if
#POP pupcommand pupout addr badsectorflag linecounter
#POP maxlines
#OUTPUT
```

Figure 8-1. Monitoring System Status (Page 11 of 12)

```
== ------------------------------------------------------
== Check the status of other devices
== ------------------------------------------------------
#PUSH puplistdev scanline state device otherdevs

#OUTPUT Checking status of other devices ...
PUP /OUTV puplistdev/ LISTDEV
#EXTRACTV puplistdev scanline == headings
#EXTRACTV puplistdev scanline == blank
#EXTRACTV puplistdev scanline == $0
[#IF [#MATCH OFF [#CHARGET scanline 18 FOR 3] ]
  |THEN| #OUTPUT $0 logging is OFF.
         #OUTPUT
] == end if

#SET otherdevs -1
[#LOOP |WHILE| NOT [#EMPTYV puplistdev] | DO |
  #EXTRACTV puplistdev scanline
  #SET device [#CHARGET scanline 65 FOR 4]
    == the #IF statement will skip:
    ==  lines with no state (device is OK)
    ==  lines with no device number (-B backups)
    ==  lines with type 3 (disk, handled in above code)
    ==  lines without a $ in device name (page banners)
  #SET state [#CHARGET scanline 18 FOR 1]
  [#IF NOT [#EMPTYV state] AND
       NOT [#EMPTYV device] AND
       NOT [#MATCH 3 [device]] AND
       [#MATCH $ [#CHARGET scanline 6 FOR 1]]
  |THEN|
    [#IF otherdevs |THEN|
      #SET otherdevs 0
      #OUTPUT [Tcr]Check the following devices:[Tce]
    ] == end if
```

Figure 8-1. Monitoring System Status (Page 12 of 12)

```
      [#CASE [state]
        |D| #OUTPUT [#CHARGET scanline 6 FOR 10] down
        |H| #OUTPUT [#CHARGET scanline 6 FOR 10] Hard down
        |R| #OUTPUT [#CHARGET scanline 6 FOR 10] reviving
        |S| #OUTPUT [#CHARGET scanline 6 FOR 10] Special state
        |I| #OUTPUT [#CHARGET scanline 6 FOR 10] Inaccessible
        |OTHERWISE|
            #OUTPUT [#CHARGET scanline 6 FOR 10] state: &
 [state]
     ] == end case
   ] == end if
] == end loop
] == end if [version] = 20

#POP puplistdev scanline state device otherdevs
#OUTPUT

TIME
#UNFRAME
```

(This page left intentionally blank)

9 Syntax Summary

The syntax diagrams summarized in this appendix are divided into five categories:

- The command interpreter set of commands and functions, supplied with TACL in the directory :UTILS:TACL
- The built-in functions and variables that constitute the TACL programming language
- The specialized forms of the #DEF function used to create and redefine structured variables (STRUCT declarations)
- The specialized forms of the #SET function used to assign values to TACL built-in variables
- The commands of the #DELTA character processor

:UTILS:TACL Commands and Functions

The following summarizes the syntax of the TACL command interpreter commands and functions:

```
ACTIVATE [ [\node-name.]{$process-name | cpu,pin } ]

ADD DEFINE { define-name
           { ( define-name [, define-name ] … ) }
   [, LIKE define-name ] [, attribute-spec ] …

ADDDSTTRANSITION start-date-time , stop-date-time , offset

ADDUSER [ / run-option [ , run-option ] … / ]
   group-name.user-name , group-id,user-id

ALARMOFF

ALTER DEFINE define-name-list { , attribute-spec
                              { , RESET reset-list }

ALTPRI [\node-name.]{$process-name | cpu,pin } , pri

ASSIGN [ logical-unit [ , [ actual-file-name ]
   [ , create-open-spec ] … ] ]

ATTACHSEG { PRIVATE } file-name directory-name
          { SHARED  }

BACKUPCPU [ cpu ]

BREAK [ variable-level ]

BUILTINS [ / { FUNCTIONS | VARIABLES } / ]

BUSCMD  [ / run-option [ , run-option ] … / ]
   { X | Y } , { DOWN | UP } , from-cpu , to-cpu
```

```
        ⎧ ALL                  ⎫
        ⎪ ALL ASSIGN           ⎪
CLEAR   ⎨ ALL PARAM            ⎬
        ⎪ ASSIGN logical-unit  ⎪
        ⎩ PARAM param-name     ⎭

COLUMNIZE list

COMMENT [ comment-text ]

_COMPAREV string-1 string-2

COMPUTE expression

_CONTIME_TO_TEXT contime-list

_CONTIME_TO_TEXT_DATE contime-list

_CONTIME_TO_TEXT_TIME contime-list

COPYDUMP [ / run-option [ , run-option ] … / ]
   source-file , dest-file

COPYVAR variable-level-in variable-level-out

CREATE file-name [ , extent-size ]

CREATESEG file-name

DEBUG [ [\node-name.]{$process-name | cpu,pin } ]
   [ , TERM [\node-name.]$terminal-name ]

_DEBUGGER current-trace-value reason-for-entry

DEFAULT [ / run-option [, run-option ] …  / ]
  {default-names [, "default-security" ]}
                 , "default-security"

DELETE DEFINE define-name-list

DELUSER [ / run-option [ , run-option ] …  / ]
   group-name.user-name

DETACHSEG directory-name

ENV [ environment-parameter ]

EXIT

   [ num  ]
FC [ -num ]
   [ text ]

FILEINFO [ / OUT list-file / ]
   [ file-name-template [ [,] file-name-template ] … ]

FILENAMES [ / OUT list-file / ]
   [ file-name-template [ [,] file-name-template ] … ]
```

```
FILES [ / OUT list-file / ]
   [ subvol-template [ [,] subvol-template ] … ]

FILETOVAR file-name variable-level

HELP

HISTORY [ num ]

HOME [ directory-name ]

INFO [ / OUT list-file / ] DEFINE define-name-list [, DETAIL ]

INITTERM

INLECHO { OFF | ON }

INLEOF

INLOUT { OFF | ON }

INLPREFIX [ prefix ]

INLTO [ variable-level ]

JOIN variable-level

KEEP [ / LIST / ] num variable [ variable ] …

KEYS

LIGHTS [ / run-option [, run-option ] .../ ]
       [ ON             ] [ , sys-option ]
       [ OFF            ] [ , ALL        ] … [, BEAT ]
       [ SMOOTH [num]   ]

LOAD [ / KEEP num / ] file-name [ file-name ] …

LOGOFF [ / option [, option ] … / ]

       { group-name.user-name }
LOGON  { group-id,user-id     }
       { alias                }
[ ,password                                  ]
[ ,old-password,^new-password                ]
[ ,old-password,^new-password,^new-password  ]
   [ ; parameter [, parameter ] ...]

_LONGEST list

_MONTH3 num

O[BEY] command-file

OUTVAR [ / option [ , option ] … / ] string

PARAM [ param-name param-value
   [ , param-name param-value ] … ]

PASSWORD [ / run-option [ , run-option ] … / ]
   [ password ]
```

```
PAUSE [ [\node-name.]{$process-name | cpu,pin } ]

PMSEARCH subvol-spec [ [,] subvol-spec ] …

PMSG { ON | OFF }

POP variable [ [,] variable ] …

PPD [ / OUT list-file / ]
   [ [\node-name.][ { $process-name | cpu,pin | * } ] ]

PURGE / option / file-name-template [, file-name-template … ]

PUSH variable [ [,] variable ] …

RCVDUMP [ / run-option [ , run-option ] … / ]
   dump-file , cpu , { X | Y } [ , param [ , param ] ]

RECEIVEDUMP / OUT dump-file / cpu , bus
   [ , param [ , param ] ]

RELOAD [ / run-option [, run-option ] … / ]
   [ cpu-set [; cpu-set ] …  ]
   [ HELP                    ]

REMOTEPASSWORD [ \node-name [ , password ] ]

RENAME old-file-name [,] new-file-name

RESET DEFINE { attribute-name [, attribute-name ] … }
             { *                                    }

[ RUN[D] ] program-file [ / run-option [ , run-option ] … / ]
   [ param-set ]

SEGINFO

SET DEFINE { attribute-spec } [, attribute-spec ] …
           { LIKE define-name }

SET DEFMODE { ON | OFF }

SET HIGHPIN { ON | OFF }

SET INSPECT { OFF | ON | SAVEABEND }

SETPROMPT { SUBVOL | VOLUME | BOTH | NONE }

SET SWAP [ $volume-name ]

SETTIME { month day } year , hour:min[:sec] [ GMT ]
        { day month }                       [ LST ]
                                            [ LCT ]

SET VARIABLE [ / option [, option ]/ ] variable-level [ text ]

SET VARIABLE built-in-variable [ built-in-text ]

SHOW [ / OUT list-file / ] [ attribute [ , attribute ] … ]
```

```
SHOW [ / OUT list-file / ] DEFINE [ attribute-name | * ]

SINK [ text ]

STATUS [ / OUT list-file / ] [ range ] [ , condition ] …
   [ , DETAIL ] [ , STOP ]

STOP [ [\node-name.]{$process-name | cpu,pin } ]

SUSPEND [ [\node-name.]{$process-name | cpu,pin } ]

SWITCH

SYSTEM [ \node-name ]

SYSTIMES

[\node-name.]TACL [ / run-option [ , run-option ] … / ]
   [ backup-cpu-num ] [ ; parameter [ , parameter ] ]

TIME

USE [ directory-name [ [,] directory-name ] … ]

USERS [ / run-option [ , run-option ] … / ] [ range ]

VARIABLES [ directory-name ]

VARINFO [ variable [ [,] variable ] … ]

VARTOFILE variable-level file-name

VCHANGE [ / option [ , option ] … / ] variable-level
   string-1 string-2 [ range ]

VCOPY [ / option [ , option ] … /  ] source-var range
   dest-var dest-line

VDELETE [ / option [ , option ] / … ] variable-level range

VFIND [ / option [ , option ] / … ] variable-level string
   [ range ]

VINSERT variable-level line-num

VLIST [ / option [ , option ] / … ] variable-level [ range ]

VMOVE [ / option [ , option ] / … ] source-var range
   dest-var dest-line

VOLUME [ [\node-name.]volume ] [ , "security" ]

VTREE [ directory-name ]

WAKEUP { ON | OFF }

WHO

{ X | Y }BUSDOWN from-cpu , to-cpu

{ X | Y }BUSUP from-cpu , to-cpu
```

```
! [ num    ]
  [ -num   ]
  [ text   ]

? [ num    ]
  [ -num   ]
  [ text   ]
```

Built-In Functions and Variables

The following summarizes the syntax of the built-in functions and variables used for programming in TACL:

```
#ABEND [ / option [ , option ] … / ]
   [ [\node-name.]{$process-name | cpu,pin } [ text ] ]

#ABORTTRANSACTION

#ACTIVATEPROCESS [ [\node-name.]{$process-name | cpu,pin} ]

#ADDDSTTRANSITION low-gmt high-gmt offset

#ALTERPRIORITY [ [\node-name.]{$process-name | cpu,pin } ]
   pri

#APPEND to-variable-level [ text ]

#APPENDV to-variable-level { from-variable-level | string }

#ARGUMENT [ / option [ , option ] … / ]
   alternative [ alternative ] …

#ASSIGN [ / option [ , option ] … / logical-unit ]

#BACKUPCPU [ cpu ]

#BEGINTRANSACTION

#BREAKMODE

#BREAKPOINT variable-level state

#BUILTINS [ / { FUNCTIONS | VARIABLES } / ]

#CASE text enclosure

#CHANGEUSER [ / CHANGEDEFAULTS / ] { group-name.user-name }
                                   { group-id,user-id     }
                                   { alias                }
  password

#CHARACTERRULES

#CHARADDR variable-level line-addr

#CHARBREAK variable-level char-addr

#CHARCOUNT variable-level

#CHARDEL variable-level char-addr-1 [ FOR char-count ]
                                    [ TO^char-addr-2 ]
```

```
#CHARFIND [ / EXACT / ] variable-level char-addr text

#CHARFINDR [ / EXACT / ] variable-level char-addr text

#CHARFINDRV [ / EXACT / ] variable-level char-addr string

#CHARFINDV [ / EXACT / ] string-1 char-addr string-2

#CHARGET variable-level char-addr-1 [ FOR˚char-count
                                      TO˚char-addr-2 ]

#CHARGETV var-1 var-2 char-addr-1 [ FOR˚char-count
                                    TO˚char-addr-2 ]

#CHARINS string char-addr text

#CHARINSV variable-level char-addr string

#COLDLOADTACL

#COMPAREV string-1 string-2

#COMPUTE expression

#COMPUTEJULIANDAYNO year month day

#COMPUTETIMESTAMP year month day hour min sec milli micro

#COMPUTETRANSID system cpu sequence crash-count

#CONTIME timestamp

#CONVERTPHANDLE { / PROCESSID / integer-string
                  / INTEGERS / process-identifier }

#CONVERTPROCESSTIME process-time

#CONVERTTIMESTAMP gmt-timestamp direction [ \node-name ]

#CREATEFILE [ / EXTENT num / ] file-name

#CREATEPROCESSNAME

#CREATEREMOTENAME \node-name

#DEBUGPROCESS [ / NOW / ]
   [\node-name.]{$process-name | cpu,pin }
      [ , TERM [\node-name.]$terminal-name ]

                  { { ALIAS   }
                  { { DELTA   }
                  { { MACRO   } enclosure     }
#DEF variable     { { ROUTINE }               }
                  { { TEXT    }               }
                  { DIRECTORY [ segment-spec ] }
                  { STRUCT structure-body      }
                  ˚

#DEFAULTS [ / option [ , option ] / ]

#DEFINEADD define-name [ flag ]
```

```
#DEFINEDELETE define-name

#DEFINEDELETEALL

#DEFINEINFO define-name

#DEFINEMODE

#DEFINENAMES define-template

#DEFINENEXTNAME [ define-name ]

#DEFINEREADATTR { define-name } { attribute-name }
                { _           } { cursor^^mode   }

#DEFINERESTORE [ / option [ , option ] … / ] buffer

#DEFINERESTOREWORK

#DEFINESAVE [ / WORK / ] define-name buffer

#DEFINESAVEWORK

#DEFINESETATTR attribute-name [ attribute-value ]

#DEFINESETLIKE define-name

#DEFINEVALIDATEWORK

#DELAY csecs

#DELTA [ / COMMANDS variable-level / ] [ text ]

#DEVICEINFO / option [ , option ] … /
   { $device-name | file-name }

#EMPTY [ text ]

#EMPTYV [ / BLANK / ] string

#EMSADDSUBJECT [ / SSID ssid / ] buffer-var
   token-id [ token-value ]

#EMSADDSUBJECTV [ / SSID ssid / ] buffer-var
   token-id source-var

#EMSGET [ / option [ , option ] … / ] buffer-var get-op

#EMSGETV [ / option [ , option ] … / ] buffer-var get-op
   result-var

#EMSINIT [ / option [ , option ] / ] buffer-var ssid
   event-number token-id [ token-value ] … ]

#EMSINITV [ / option [ , option ] / ] buffer-var ssid
   event-number token-id source-var

#EMSTEXT [ / option [ , option ] … / ] buffer-var

#EMSTEXTV [ / option [ , option ] … / ] buffer-var
   formatted-var [ lengths-var ]
```

```
#ENDTRANSACTION

#EOF variable-level

#ERRORNUMBERS

#ERRORTEXT  / option [ option ] … /

#EXCEPTION

#EXIT

#EXTRACT variable-level

#EXTRACTV from-variable-level to-variable-level

#FILEGETLOCKINFO [ / option / ] fvname control lockdesc
   participants

#FILEINFO / option [ , option ] … / file-name

#FILENAMES [ / option [ , option ] … / ]
   [ file-name-template ]

#FILTER [ exception [ exception ] … ]

#FRAME

#GETCONFIGURATION / option [ , option ] … /

#GETPROCESSSTATE [ / option [ , option ] ... / ]

#GETSCAN

#HELPKEY

#HIGHPIN

#HISTORY [ / option [ , option ] … / ]

#HOME

#IF [ NOT ] numeric-expression [ enclosure ]

#IN

#INFORMAT

#INITTERM

#INLINEECHO

#INLINEEOF

#INLINEOUT

#INLINEPREFIX

#INLINEPROCESS

#INLINETO

#INPUT [ / option [ , option ] … / ] [ prompt ]
```

```
#INPUTEOF

#INPUTV [ / option [ , option ] … / ] variable-level
   prompt-string

#INSPECT

#INTERACTIVE  [ / CURRENT / ]

#INTERPRETJULIANDAYNO julian-day-num

#INTERPRETTIMESTAMP four-word-timestamp

#INTERPRETTRANSID transid

#JULIANTIMESTAMP [ type [ tuid-request ] ]

#KEEP num variable

#KEYS

#LINEADDR variable-level char-addr

#LINEBREAK variable-level line-addr char-offset

#LINECOUNT variable-level

#LINEDEL variable-level line-addr-1 [ FOR line-count
                                    TO line-addr-2 ]

#LINEFIND [ / EXACT / ] variable-level line-addr text

#LINEFINDR [ / EXACT / ] variable-level line-addr text

#LINEFINDRV [ / EXACT / ] variable-level line-addr string

#LINEFINDV [ / EXACT / ] variable-level line-addr string

#LINEGET string line-addr-1 [ FOR line-count
                            TO line-addr-2 ]

#LINEGETV string variable-level line-addr-1 [ FOR line-count
                                            TO line-addr-2 ]

#LINEINS variable-level line-addr text

#LINEINSV variable-level line-addr string

#LINEJOIN variable-level line-addr

#LOAD [ / option [ , option ] / ] file-name

#LOCKINFO lock-spec tag buffer

#LOGOFF [ / option [ , option ] … / ]

#LOOKUPPROCESS / option [ , option ] … / specifier

#LOOP enclosure

#MATCH template [ text ]
```

```
#MOM

#MORE

#MYGMOM

#MYPID

#MYSYSTEM

#MYTERM

#NEWPROCESS program-file [ / option [, option ]… / ]
   [ param-set ]

#NEXTFILENAME [ file-name ]

#OPENINFO / option [ , option ] / { file-name | device-name }
   tag

#OUT

#OUTFORMAT

#OUTPUT [ / option [ , option ] … / ] [ text ]

#OUTPUTV [ / option [ , option ] … / ] string

#PARAM [ param-name ]

#PAUSE [ [\node-name.]{$process-name | cpu,pin } ]

#PMSEARCHLIST

#PMSG

#POP variable [ [,] variable ] …

#PREFIX

#PROCESS

#PROCESSEXISTS [\node-name.]{$process-name | cpu,pin }

#PROCESSFILESECURITY

#PROCESSINFO / option [ , option ] … /
   [ [\node-name.]{$process-name | cpu,pin } ]

#PROCESSORSTATUS [ \node-name ]

#PROCESSORTYPE [ / BOTH | NAME / ]
   { [\node-name.]{$process-name | cpu,pin } }
   { cpu-num                                 }
#PROMPT

#PURGE file-name

#PUSH variable [ [,] variable ] …

#RAISE exception
```

```
#RENAME old-file-name new-file-name

#REPLY [ text ]

#REPLYPREFIX

#REPLYV string

#REQUESTER [ / option [ , option ] / ]
   { CLOSE variable-level                                  }
   { READ file-name^error-var^read-var^prompt-var          }
   { WRITE file-name^error-var^write-var                   }

#RESET option [ option ] …

#REST

#RESULT [ text ]

#RETURN

#ROUTINENAME

#SEGMENT [ / USED / ]

#SEGMENTCONVERT / FORMAT { a | b } / old-file-name
   new-file-name

#SEGMENTINFO / option [ , option ] / [ segment-id ]

#SEGMENTVERSION file-name

#SERVER / option [ , option ] … / [ server-name ]

#SET {[ / option [, option ] / ] variable-level [ text ]}
     {built-in-variable [ built-in-text ]               }

#SETBYTES destination source

#SETCONFIGURATION / option [, option ] .../[ tacl-image-name]

#SETMANY variable-name-list , [ text ]

#SETPROCESSSTATE

                      { LOGGEDON             }
                      { TSNLOGON             }
#SETPROCESSSTATE / {  { STOPONLOGOFF         } / { 0 | 1 }
                      { PROPAGATELOGON       }
                      { PROPAGATESTOPONLOGOFF }

#SETSCAN num

#SETSYSTEMCLOCK julian-gmt mode [ tuid ]

#SETV dest-variable-level source-string

#SHIFTDEFAULT

#SHIFTSTRING [ / option / ] [ text ]
```

```
#SORT [ / option / ] [ text ]

#SPIFORMATCLOSE

#SSGET [ / option [ , option ] … / ] buffer-var get-op

#SSGETV [ / option [ , option ] … / ] buffer-var get-op
   result-var

#SSINIT [ / TYPE 0 / ] buffer-var ssid command
   [ / type-0-option [ , type-0-option ] … / ]

#SSMOVE [ / option [ , option ] … / ] source-var dest-var
   token-id

#SSNULL token-map struct

#SSPUT [ / option [ , option ] … / ] buffer-var token-id
   [ token-value [ token-value ] … ]

#SSPUTV [ / option [ , option ]… / ] buffer-var token-id
   source-var

#STOP [ / option [ , option ] … / ]
   [ [\node-name.]{$process-name | cpu,pin } [ text ] ]

#SUSPENDPROCESS [ [\node-name.]{$process-name | cpu,pin} ]

#SWITCH

#SYSTEM [ \node-name ]

#SYSTEMNAME system-number

#SYSTEMNUMBER \node-name

#TACLOPERATION

#TACLSECURITY

#TACLVERSION / REVISION /

#TIMESTAMP

#TOSVERSION [ \node-name ]

#TRACE

#UNFRAME

#USELIST

#USERID user

#USERNAME user

#VARIABLEINFO / option [ , option ] … / variable-level

#VARIABLES [ / { BREAKPOINT | IO } / ]

#VARIABLESV [ / { BREAKPOINT | IO } / ] variable-level
```

```
#WAIT variable-level [ variable-level ] …
#WAKEUP
#WIDTH
```

STRUCT Declarations

The following summarizes the forms of the #DEF function used to create and redefine structured variables:

```
#DEF variable STRUCT
   { BEGIN declaration [ declaration ] … END ; }
   { LIKE structure-identifier ;               }

type identifier [ VALUE initial-value ] ;

type identifier ( lower-bound : upper-bound )
   [ VALUE initial-value ] ;

STRUCT identifier [ ( lower-bound : upper-bound ) ] ;
   { BEGIN declaration [ declaration ] … END ; }
   { LIKE structure-identifier ;               }

FILLER num ;

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

#SET Summary

The following summarizes the syntax of the #SET function when it is used to assign values to built-in variables. SET VARIABLE commands used for the same purpose have the same syntax.

```
#SET #ASSIGN [ [ / option [ , option ] … / ] logical-unit ]
#SET #BREAKMODE { DISABLE | ENABLE | POSTPONE }
#SET #CHARACTERRULES file-name
#SET #DEFAULTS subvolume-name
#SET #DEFINEMODE { OFF | ON }
#SET #ERRORNUMBERS n n n n
#SET #EXIT num
#SET #HELPKEY [ key-name ]
#SET #HIGHPIN { OFF | ON }
#SET #HOME directory
#SET #IN file-name
#SET #INFORMAT { PLAIN | QUOTED | TACL }
#SET #INLINEECHO num
#SET #INLINEOUT num
#SET #INLINEPREFIX [ prefix ]
#SET #INLINETO [ variable-level ]
#SET #INPUTEOF num
#SET #INSPECT { OFF | ON | SAVEABEND }
#SET #MYTERM home-term
#SET #OUT file-name
#SET #OUTFORMAT { PLAIN | PRETTY | TACL }
#SET #PARAM [ param-name [ param-value ] ]
#SET #PMSEARCHLIST searchlist
#SET #PMSG num
#SET #PREFIX [ text ]
#SET #PROCESSFILESECURITY "security"
#SET #PROMPT num
#SET #REPLYPREFIX [ num]
#SET #SHIFTDEFAULT { DOWN | NOOP | UP }
#SET #TACLSECURITY "security"
#SET #TRACE num
#SET #USELIST [ directory-name [ directory-name ] … ]
```

```
#SET #WAKEUP num
#SET #WIDTH num
```

#DELTA Command Summary

Table 9-1 summarizes the syntax of the #DELTA character processor commands.

Table 9-1. #DELTA Commands (Page 1 of 2)

Command	Description
*x*A	Convert ASCII
y,*x*A	Convert ASCII with error return
B	Beginning
*x*C	Character move
x:C	Character move with return code
*x*D	Delete
EI*file*$	Open file for input
EO*file*$	Open file for output
*x*FC	Lowercase lines
y,*x*FC	Lowercase characters
x@FC	Uppercase lines
y,*x*@FC	Uppercase characters
FE*var*$	Test variable level for emptiness
FF*var*$	Get frame number of variable level
*x*FG*var*$	Compare lines to variable level
y,*x*FGvar$	Compare range to variable level
FL	Get length from last I or S operation
FO*var*$	Pop variable
FT*var*$	Get variable type
*x*FT*var*$	Set variable type
FU*var*$	Push variable
*x*FU*var*$	Push and load variable with *x*
G*var*$	Get text from variable level
H	Whole buffer
I*text*$	Insert text
*x*I	Insert ASCII
y,*x*I	Insert *y**ASCII
*x*J	Jump characters
*x*K	Kill lines
y,*x*K	Kill characters
*x*L	Move by lines
M*var*$	Invoke macro

Table 9-1. #DELTA Commands (Page 2 of 2)

Command	Description
*x*P	Write lines
y,*x*P	Write characters
Q*var*$	Get value from variable level
*x*S*text*$	Search
x:S*text*$	Search with return code
*x*T	Type lines
y,*x*T	Type characters
@T*var*$	Type variable level contents
:T*text*$	Type text
*x*U*var*$	Unload *x* into variable level
y,*x*U*var*$	Unload *x* into variable level
*x*V	View lines
x:V	View lines and show end
*x*X*var*$	Extract lines to variable level
y,*x*X*var*$	Extract characters to variable level
*x*Y	Read lines
Z	Get buffer size
\	Convert number in text to value in X
x\	Put *x* in text
^\	Exit from macro
?	Condition
:?	NOT condition
'	End condition
,	Move X into Y
$	Clear X and Y
.	Get current position
=	Display X or Y,X
<	Begin iteration
x<	Iterate *x* times
;	Exit iteration
>	End iteration
@>	End iteration, do not decrement iteration count
!	Comment

Appendix A Supplemental Information for D-Series Systems

The D-series operating system supports expanded numbers of processes and devices. This appendix introduces the features available in D-series software and includes the following information:

- ☐ An overview of D-series operating system features
- ☐ Changes to the $CMON interface
- ☐ Changes to completion code handling
- ☐ Changes to the EMS and SPI interfaces

For detailed information about D-series changes to the TACL product, including a list of required and optional changes, see the *TACL Reference Manual*.

The following manuals contain additional information about D-series software:

- ☐ The *Introduction to D-series Systems* provides an overview of D-series enhancements.
- ☐ The *Guardian Application Conversion Guide* describes differences between C-series and D-series applications.
- ☐ The *Guardian Programmer's Guide* provides programming information for a D-series operating system.
- ☐ The *System Procedure Calls Reference Manual* describes syntax and programming considerations for D-series procedures.
- ☐ The *System Procedure Errors and Messages Manual* describes error codes, system messages, and trap numbers for C-series and D-series procedures.
- ☐ The *D-series System Migration Planning Guide* contains planning information for migration from a C-series operating system to a D-series operating system.

D-Series Operating System Features

The D-series operating system provides the following features:

- ☐ More concurrent processes per CPU.

  The D-series operating system allows more than 256 concurrent processes per CPU (C-series software is limited to 256 concurrent processes per CPU). The actual usable number of processes depends on available resources such as physical memory.

- ☐ More devices per system.
- ☐ New file name storage format.

  The D-series file name is stored as a variable-length string. The new storage format applies to all file names, including disk files, devices, and processes.

- ☐ New process identifiers.

  The range of values for process identification numbers (PINs) has been expanded. The D-series process file name, a variable-length string, replaces the C-series process file name. The process handle is a new ten-word value that replaces the C-series four-word process identifier. TACL supports a new field type for STRUCT variables, called PHANDLE.

- ☐ New system procedures.

  D-series procedures can accept and return longer file names and process names, and larger PIN values; new system procedures support these capabilities.

- ☐ New system messages.

  These messages support the enhanced capability of the D-series operating system.

- ☐ New object file attributes.

  There is a new RUN attribute, HIGHPIN, available for program files.

In general, D-series changes are available as options.

Influence on Examples in This Manual

Examples listed in this manual run on D-series software without modification. Refer to the guidelines in Section 1, "An Overview of TACL," for steps to perform before running the examples.

Communicating With a $CMON Process

Section 5, "Initiating and Communicating With Processes," contains an example that illustrates communication with a $CMON process. If you communicate with a $CMON process and run processes at high PINs, note that the format of the ALTERPRIORITY message has been extended to support process handles.

When a user attempts to alter the priority of a process, the TACL process sends an ALTERPRIORITY (-56) message to $CMON. This message contains the process identifier of the target process.

Not all D-series processes can be represented by a C-series process identifier (CRTPID); therefore, the ALTERPRIORITY (-56) message has been extended to include the process handle of the target process. If the process handle can be converted to a process identifier, TACL also includes the process identifier; otherwise, TACL sets the process identifier field to zero.

The ALTERPRIORITY message is defined as follows:

```
STRUCT altpri^msg;
 BEGIN
  INT   msgcode;          == -56
  INT   userid;           == user altering the priority
  INT   cipri;            == command interpreter
                          ==   initial priority
  INT   ciinfile [0:11];  == command interpreter IN file
  INT   cioutfile [0:11]; == command interpreter OUT file
  INT   crtpid [0:3];     == process identifier of target
                          ==   process
  INT   progname [0:11];  == program file of target process
  INT   priority;         == new priority for target process
  INT   phandle [0:9];    == phandle of target process
 END;
```

TACL sends the new message to a C-series $CMON if a user on a D-series operating system attempts to alter the priority of a process running on a C-series system. The C-series $CMON process can extract a valid process identifier and ignore the extra words in the message. The process handle for a process running on a C-series system can always be converted to a process identifier.

Processing Completion Information

Section 5, “Initiating and Communicating With Processes,” describes how to use the variable:_COMPLETION to access completion information.

Because a D-series PIN does not fit into a CRTPID field, D-series TACL uses a new structure for completion information. The TACL product provides C-series compatibility by continuing to support the use of :_COMPLETION.

In previous releases, TACL saved STOP (-5) and ABEND (-6) messages in the variable :_COMPLETION, if it existed. (TACL defines :_COMPLETION as a STRUCT when you log on, and the STRUCT remains unless you pop it.)

D-series TACL receives PROCDEATH (-101) messages instead of STOP and ABEND messages. TACL saves a PROCDEATH message in the variable :_COMPLETION^PROCDEATH, if it exists. (TACL defines :_COMPLETION^PROCDEATH as a STRUCT when you log on, and the STRUCT remains unless you pop it.)

If the variable :_COMPLETION exists, TACL converts the PROCDEATH system message to a C-series STOP or ABEND system message and stores the message in :_COMPLETION. Note, however, that if the message represents an unnamed high PIN process, the message will not fit in :_COMPLETION. In such a case, TACL fills :_COMPLETION with zeroes.

New D-series TACL applications should use :_COMPLETION^PROCDEATH. Existing TACL applications may continue to use :_COMPLETION; however, the PIN in the process identifier field is set to 255 for any high PIN value.

TACLSEGF (supplied by Tandem) defines :_COMPLETION^PROCDEATH as follows:

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

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

You can access the process name as follows:

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

You can access the termination text as follows:

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

Communicating With Programmatic Interfaces

Section 7, "Using Programmatic Interfaces," describes how to access the SPI and EMS interfaces. To describe a high-PIN process to one of these interfaces, you must use a process handle in place of a process identifier (CRTPID). The D-series TACL product recognizes a new data type called PHANDLE (process handle) for STRUCT variables.

TACL uses ten unsigned integers, separated by periods, to represent a process handle in external form (as returned by #SSGET or #OUTVAR). Each integer can range from 0 to 65535. Use this external form whenever you send a process handle to TACL (as used by #SSPUT and #SET). The following example shows a process handle in TACL external form:

```
1.3.5.7.9.11.13.15.17.19
```

To display a process handle, you can use the OUTVAR command or #OUTPUTV built-in function. The #VARIABLEINFO built-in function with option TYPE returns type PHANDLE for a process handle field in a STRUCT.

#SSGET and #SSPUT convert a process handle to and from the external form. Neither function checks the validity of the handle, but #SSPUT checks to make sure the handle contains ten unsigned integers, each with a value between zero and 65535.

D-series TACL supports the new token data type ZSPI^TDT^PHANDLE, which has the value type of a process handle. The format of ZSPI^TDT^PHANDLE consists of

ten integers, each ranging from 0 to 65535. A null process handle consists of ten integers, each of which has the value 65535.

A new built-in function, #CONVERTPHANDLE, converts process file identifiers to process handles and process handles back to process descriptors.

A new built-in function, #SPIFORMATCLOSE, closes the template file defined in =_EMS_TEMPLATES so that you can open a new template file.

Glossary

access mode. A file attribute that determines what operations you can perform on the file, like reading and writing.

alias. An alternative name for a given function.

ancestor. The process that is notified when a named process or process pair is deleted. The ancestor is usually the process that created the named process or process pair.

argument. A parameter that you specify when you invoke a macro or routine.

array data item. A portion of a STRUCT that is treated as an array; that is, you can refer to the whole item, or you can refer to individual elements of it.

ASSIGN. An association of a physical file name with a logical file name made by the TACL ASSIGN command. The physical file name is any valid file name. The logical file name is used within a program. The ASSIGN is therefore used to pass file names to programs.

BREAK mode. A mode of process execution where a process gains exclusive access to a terminal when the BREAK key is pressed. BREAK mode is established using SETPARAM function 3 or SETMODE function 11.

BREAK owner. The process that receives the break-on-device message when the BREAK key is pressed. The establishment of BREAK ownership is achieved using SETPARAM function 3 or SETMODE function 11.

breakpoint. A location (or point) in a program where execution is to be suspended so that you can then examine and perhaps modify the state of the program. You can set and clear breakpoints with _DEBUGGER commands.

built-in. A function or variable built into TACL; a built-in cannot be modified. Other variables can be modified by the user.

C-series system. A system that is running a C-series version of the Guardian 90 operating system.

CAID. See creator access ID.

child process. A process created by the current process.

code segment. An area of memory that contains program instructions to be executed, plus related information. An absolute segment whose logical pages are read from but never written back to the swap file.

command. A text string that directs the computer to perform a task. Commands are usually composed of a verb that tells the computer what to do and an object or list of objects that is acted on by the verb. TACL commands are interpreted by TACL and are extensible.

command-interpreter monitor ($CMON). A server process that monitors requests made to the TACL process and affects the way TACL responds.

completion code. A value used to return information about a process to its ancestor process when the process is deleted. This value is returned in the process deletion message, system message -101.

condition code. A status returned by some file-system procedure calls to indicate whether the call was successful. A condition-code-greater-than (CCG) indicates a warning, a condition-code-less-than (CCL) indicates an error, and a condition-code-equal (=) indicates successful execution.

conversational mode. A mode of communication between a terminal and its I/O process in which each byte is transferred from the terminal to the processor I/O buffer as it is typed. Each file-transfer operation finishes when a line-termination character is typed at the terminal. Contrast with page mode.

creator. The process that initiates execution of another process. Compare with mom and ancestor.

creator access ID (CAID). A process attribute that identifies, by user ID, the user who initiated the process creation. Contrast with process access ID.

data segment. A type of absolute segment whose logical pages contain information to be processed by the instructions in the related code segment.

deadlock. A situation in which two processes or two transactions cannot continue because they are each waiting for the other to release a lock.

default process. The process whose name is returned by the #PROCESS function. It is the process most recently created by a RUN or RUND command, an implied RUN, or a #NEWPROCESS built-in function, or for which TACL was most recently paused by a PAUSE *proc-spec* command or a #PAUSE *proc-spec* built-in function; if that process is no longer running, there is no default process.

DEFINE. A named set of attributes and values.

DEFINE name. An identifier preceded by an equal sign that can be used in place of an actual name to identify a DEFINE in a procedure call. See DEFINE.

Delta. The low-level character editor provided by TACL.

destination control table (DCT). A collection of operating system data structures that serves as a directory of named processes and logical devices.

device. A peripheral hardware attachment used for input and output; for example, a printer or a disk.

device subtype. A value that further qualifies a device type. For example, a device type of 4 indicates a magnetic tape drive; if the same device has a device subtype of 2, then the magnetic tape drive has a 3206 controller.

disk volume. Also called a disk or a volume; a magnetic storage medium. Disk names consist of a dollar sign ($) followed by one to seven alphanumeric characters (network) or one to eight alphanumeric characters (local), the first of which must be alphabetic.

EDIT file. A file in a format defined by the EDIT product.

enclosure. A unit composed of one or more labels, such as |THEN| or |DO|, and the text associated with each label. Enclosures are found only in the TACL built-in functions #DEF, #IF, #CASE, and #LOOP, which are enclosed in brackets to provide boundaries for their enclosures. TACL defers execution of text that is associated with labels until it determines the correct label to use.

Enscribe. A database record management system.

exception. An unusual event that causes TACL to interrupt the normal flow of invocations and transfer to special code (see exception handler). This unusual event could be BREAK, a TACL error, or a user-defined exception.

exception handler. A series of TACL statements that perform resource deallocation and cleanup after an exception.

exclusion mode. The attribute of a lock that determines whether any process except the lock holder can access the locked data.

expand. A type of invocation (see invoke). To expand a variable, specify the variable name in brackets; TACL returns the expansion in place of the variable name.

expression. A text, string, or integer constant, a variable, or a value obtained by combining constants, variables, and other expressions with operators. Expressions are used as arguments to commands and built-in functions.

extended data segment. One or more consecutive absolute segments that are dynamically allocated by a process.

extensible data segment. An extended data segment for which swap file extents are not allocated until needed.

extent. A contiguous area of a disk allocated to the same file.

fault tolerance. The ability of a computer system to continue operating during and after a fault (the failure of a system component).

file code. An integer value assigned to a file for application-dependent purposes.

file lock. A mechanism that restricts access to a file by all processes except the lock owner.

file. As used here, a file refers to an organized collection of data stored on a disk. In general, a file on a Tandem system can be a disk file, a process, or a device.

file name. A unique name for a file. This name is used to open a file and thereby provides a connection between the opening process and the file. File names consist of one to eight alphanumeric characters, the first of which must be alphabetic.

file name template. A sequence of characters including the asterisk (*) and question mark (?) that matches existing file names by expanding each asterisk to zero or more letters, digits, dollar signs ($), and pound signs (#) and replacing each question mark with exactly one letter, digit, dollar sign, or pound sign.

file system. A set of operating system procedures and data structures that provides for communication between a process and a file, which can be a disk file, a device other than a disk, or another process.

FILLER byte. A portion of a STRUCT that is used only to maintain the alignment of adjacent STRUCT items.

frame. A local environment managed by the #FRAME, #UNFRAME, and #RESET built-in functions.

fully qualified file name. The complete name of a file, including the node name. For permanent disk files, this file name consists of a node name, volume name, subvolume name, and file ID. For temporary disk files, the file name consists of a node name, a subvolume name, and a temporary file ID. For a device, the file name consists of a node name and a device name or logical device number. For a named process, the file name consists of a node name, and a process name. For an unnamed process, the file name consists of a node name, CPU number, PIN, and sequence number. Contrast with partially qualified file name.

function. An operation or set of operations that is invoked by the appearance of the function name and its arguments at the point where the result of the function is wanted. A built-in function is hard coded into TACL; users can define other functions. Variable types for functions include TEXT, MACRO, and ROUTINE.

GMT. See Greenwich mean time.

godmother. See job ancestor.

Greenwich mean time (GMT). The mean solar time for the meridian at Greenwich, England.

Gregorian date. A date specified according to the common calendar using the month of the year (January through December), the day of the month, and the year A.D.

home terminal. The terminal whose name is returned by a call to the MYTERM procedure, or the name returned in the *hometerm* parameter of the PROCESS_GETINFO_ procedure. The home terminal is often the terminal from which the process or process's ancestor was started.

interprocess communication (IPC). The exchange of messages between processes in a system or network.

interrupt. The mechanism by which a processor module is notified of an asynchronous event that requires immediate processing.

invoke. A request to execute TACL code. To invoke a variable, (1) list its name (like an implied RUN statement) without regard to results or (2) surround the variable name in square brackets ([]) to replace the name with its expansion (text or macro variable) or results (routine).

IPC. See interprocess communication.

job ancestor. A process that is notified when a process that is part of a job is deleted. The job ancestor of a process is the process that created the job to which the process belongs.

Julian timestamp. The number of microseconds since midnight January 1, 4713 B.C. at the Greenwich meridian.

LCT. See local civil time.

LDEV. See logical device.

level. One element of the set of values stored in a stack and known as a variable.

local civil time (LCT). Wall-clock time in the current time zone, including any compensation for daylight-saving time.

local standard time (LST). The time of day in the local time zone excluding any compensation made for daylight-saving time.

logical device (LDEV). (1) An addressable device, independent of its physical environment. Portions of the same logical device can be located in different physical devices, or several logical devices or parts of logical devices can be located in one physical device. (2) A process that can be accessed as if it were an I/O device; for example, the operator process is logical device LDEVOPR.

logical device number. A number that identifies a configured logical device. A logical device number can be used instead of a device file name when opening a device file.

LST. See local standard time.

macro. A named sequence of one or more instructions invoked by the appearance of the macro name. When a macro is invoked, TACL replaces arguments of the form %n% with actual arguments passed to it and returns, as a result, the instructions that define the macro, including argument values.

message system. A set of operating system procedures and data structures that handles the mechanics of exchanging messages between processes.

metacharacter. A character that directs TACL to evaluate subsequent text in a special way.

mom. A process that is notified when certain other processes are deleted. When a process is part of a process pair, the mom of the process is the other member of the pair. When a process is unnamed, its mom is usually the process that created it.

monitor. A process that, among other functions, is responsible for checking that certain other processes continue to run. If a process should stop, it is the monitor's responsibility to restart it.

multibyte character set. A means for identifying written characters for national languages that require more than one byte to represent a single character.

named process. A process to which a process name was assigned when the process was created. Contrast with unnamed process.

node. A system of one or more processor modules. Typically, a node is linked with other nodes to form a network.

node name. The portion of a file name that identifies the system through which the file can be accessed.

nonretryable error. An error condition returned by the file system that cannot be recovered by retrying the operation even after operator intervention. Contrast with retryable error.

NonStop SQL. A relational database management system that provides efficient online access to large distributed databases.

nowait I/O. An operation with an I/O device where the process does not wait for the I/O operation to finish. Contrast with waited I/O.

one-way communication. A form of interprocess communication where the sender of a message (the requester) does not expect any data in the reply from the receiver of the message (the server). Contrast with two-way communication.

operator. Perform mathematical or logical operations on values.

page. In Tandem NonStop systems, 1,024 words of contiguous data.

page mode. A mode of communication between a terminal and its I/O process in which the terminal stores up to a full page of data (1,920 bytes) in its own memory before sending the page to the I/O process. Contrast with conversational mode.

PAID. See process access ID.

PARAM. An association of an ASCII value with a parameter name made by the TACL PARAM command. You can use PARAMs to pass parameter values to processes.

partially qualified file name. A file name in which only the right-hand file name parts are specified. The remaining parts of the file name assume default values. Contrast with fully qualified file name.

partitioned file. A logical file made up of several partitions that can reside on different disks. Generic key values determine the partition on which a given record resides.

permanent disk file. A file that remains on disk until it is explicitly purged.

PFS. See process file segment.

PIN. See process identification number.

primary extent. The first contiguous area of disk allocated to a file. See also secondary extent.

priority. An indication of the precedence with which a process gains access to the instruction processing unit.

process. A program that has been submitted to the operating system for execution.

process access ID (PAID). A user ID used to determine whether a process can make requests to the system; for example, to open a file, stop another process, and so on. The process access ID is usually the same as the creator access ID, but it can be different; the owner of the corresponding object file can set the object file security such that it runs with a process access ID equal to the user ID of the file owner, rather than the creator of the process. Contrast with creator access ID.

process file name. A file name that identifies a process.

process file segment. (PFS) An extended data segment that is automatically allocated to every process and contains operating system data structures, file-system data structures, and memory-management pool data structures.

process ID. A C-series structure that serves as an address of a process.

process identification number (PIN). An unsigned integer that identifies a process in a processor module.

process name. A name that can be assigned to a process when the process is created. A process name uniquely identifies a process or process pair in a system. A process name consists of a dollar sign ($), followed by one to five alphanumeric characters, the first of which must be alphabetic.

process pair. Two processes created from the same object file running in a way that makes one process a backup process of the other in case of failure. Periodic checkpointing ensures that the backup process is always ready to take over from the primary if the primary process should fail. The process pair has one process name, but each process has a different process identification number (PIN).

process qualifier. A suffix to a process file name that gets passed to a process when the process is opened; its use is application-dependent.

process time. The amount of time a process has been active while the processor module was in the environment of the process.

processor clock. A hardware timer on each processor module that keeps processor time; the number of microseconds since cold load.

processor time. The time represented by a processor clock.

program. A sequence of instructions and data. In TACL, variables of type TEXT, MACRO, and ROUTINE can define programs.

real time. See wall-clock time.

record lock. A lock held by a process or a transaction that restricts access to that record by other processes.

redefinition. A STRUCT declaration that gives a new definition, such as a different data type or a different alignment, to an existing STRUCT or STRUCT item. All definitions are valid concurrently, allowing a STRUCT or STRUCT item to be used in a variety of ways.

reply. A response to a requester process by a server process. Contrast with request.

request. A message formatted and sent to a server by a requester. Requests also include status messages such as CPU up and CPU down messages, which are placed on the intended recipient's process message queue ($RECEIVE file) by the operating system. Contrast with reply.

requester. A process that initiates interprocess communication by sending a request to another process. Contrast with server.

response. See reply.

retryable error. An error condition returned by the file system that can be corrected by repeating the operation that caused the error. Sometimes operator intervention is required before the retry; for example, to put paper into an empty printer. Contrast with nonretryable error.

secondary extent. A contiguous area of disk storage allocated to a file. A file is made up of one or more extents; the first extent is the primary extent, and other extents are secondary extents. The secondary extents are all the same size for a specific file; the primary extent can be a different size. See also primary extent.

segment. A unit of storage consisting of up to 64 pages of 1,024 words each.

segment file. As used by TACL, a file accessible by TACL that can contain TACL code and data.

server. The process that receives, acts upon, and replies to messages from requesters. Contrast with requester.

shared data segment. An extended data segment that can be accessed by more than one process.

simple data item. A STRUCT item that contains a single value of a specific type.

space-separated list. A list whose entries are separated from each other by a space. Several built-in functions accept space-separated lists of values.

startup sequence. A convention for sending and receiving certain messages while starting a new process. By convention, the new process receives an Open message, followed by a startup message, an assign message for each ASSIGN in effect, a param message if there are any PARAMs in effect, and then a Close message.

string. A type of argument that some commands and functions accept in place of a variable. A string can be the name of a variable, text enclosed in quotation marks, or a concatenation of such entities. The concatenation operator is '+' (the single quotes are part of the operator). Under control of the QUOTED input format, a quoted string can contain TACL metacharacters.

STRUCT. A variable that is structured into individual components that can be accessed individually. Items within a STRUCT can be simple data items, arrays (which can be further broken down into individual elements), or substructures.

STRUCT item. An element of a structure that can be individually accessed by a name of the form *structure-name*:*item-name*.

substructure. A STRUCT item that is itself a STRUCT.

subvolume. A group of files stored on disk. These files all have the same subvolume name, but each has a different file ID. A subvolume name consist of one to eight alphanumeric characters, the first of which must be alphabetic. An example of a subvolume name is $DATA.INFO. An example of a file name in this subvolume is $DATA.INFO.RESULTS.

swapping. The process of copying information between physical memory and disk storage.

system process. A process whose primary purpose is to manage system resources rather than to solve a user's problem. A system process is essential to a system-provided service. Failure of a system process often causes the processor module to fail. Most system processes are automatically created when the processor module is cold loaded. Contrast with user process.

system time. The time represented by any synchronized processor clock in the system.

template. A string of characters, including the special characters * and ?, used to match another string of characters. Templates can be used in place of file names and DEFINE names in some commands and built-in functions.

temporary disk file. A file stored on disk that will be purged automatically as soon as the process that created it stops.

terminal-simulation process. A process that is made to behave like a terminal file.

text. A set of characters from the ISO 8859.1 character set. The length of text can be limited by a specific function or command. TACL interprets a text argument as all remaining text on the line, with leading and trailing spaces and end-of-line characters removed.

timekeeping. A function performed by the operating system that involves initializing and maintaining the correct time in a processor module.

timestamp. An item containing a representation of the time. A timestamp can be applied to an object at a critical point, such as the last modification time of a file.

transaction identifier. A four-word identifier that uniquely identifies a transaction within the Transaction Monitoring Facility (TMF).

transfer mode. The protocol by which data is transferred between a terminal and the computer system. See conversational mode and page mode.

two-way communication. A form of interprocess communication in which the sender of a message (requester process) expects data in the reply from the receiver (server process). Contrast with one-way communication.

unnamed process. A process to which a process name was not assigned when the process was created. Contrast with named process.

user ID. A unique pair of numbers that identify a user. A user ID has the form *group-id,user-id*, where the *group-id* identifies the user's group, and *user-id* identifies the user within the group.

user process. A process whose primary purpose is to solve a user's problem. A user process is not essential to the availability of a processor module and is created only when the user explicitly creates it. Contrast with system process.

variable. A named quantity that can assume any of a given set of values.

variable level. A portion of a variable that can be individually addressed. New levels can be added to the top of a variable stack, pushing down earlier levels, and can be popped off the top of the stack. When the last level is popped, the variable ceases to exist. For simplicity, variable levels are referred to as variables in many descriptions in this manual.

variable line. A portion of a variable level that ends with a binary zero (an internal end-of-line character). Lines can be removed from the beginning of a variable level with the #EXTRACT and #EXTRACTV functions and can be added to the end of a variable level with the #APPEND and #APPENDV function.

variable type. The designation (MACRO, DELTA, STRUCT, TEXT, and so on) of a variable level that describes its contents and the use for which it is designated.

virtual memory. A range of addresses that processes use to reference real storage, where real storage consists of physical memory and disk storage.

volume. A disk drive or a pair of disk drives that forms a mirrored disk.

waited I/O. An operation with an I/O device where the process waits until the operation finishes. Contrast with nowait I/O.

waiting process. A process that cannot execute until an event occurs, a resource becomes available, or an interval of time passes.

wall-clock time. The local time of day, including any adjustment for daylight-saving time.

working set. A collection of DEFINE attributes that have been assigned values.

$CMON. See command-interpreter monitor.

$RECEIVE. A special file name through which a process receives messages from other processes.

Index

A

Accessing structured data
Address, character 2-11
ALIAS variables 2-24
Argrec example 3-17
Arguments
  alternatives (such as NUMBER and STRING) 3-3
  parsing for a caller 3-11
  processing a space-separated list 3-8
  processing file names 3-7
  processing variables 3-8
  to routines 3-1
ASSIGN command 5-3
ATTRIBUTENAME argument alternative 3-3

B

BREAK, for debugging 2-32
Bubble example 2-3
Built-in functions
  for data manipulation 2-16
  for EMS 7-12
  for string manipulation 2-10
  #APPEND 2-3, 2-17, 4-2, 5-1, 5-5, 5-16, 5-31
  #APPEND(V) 7-3
  #APPENDV 5-37
  #ARGUMENT 3-1
  #BREAKMODE 3-30
  #CASE 2-4, 2-34, 3-25
  #CHANGEUSER 2-28
  #CHARCOUNT 2-11, 3-10
  #CHARFINDV 2-12, 3-10
  #CHARGET 2-12, 2-28, 3-9
  #CHARINSV 3-11
  #COMPAREV 4-12
  #COMPUTE 2-12, 2-22, 4-16
  #CONTIME 2-5, 4-16
  #DEF 3-5, 3-22, 7-5
  #DEFINERESTORE 5-4
  #DEFINESAVE 5-4
  #EMPTY 2-7, 3-14, 4-16
  #EMPTYV 2-7, 2-30, 5-19

Built-in functions (continued)
#EMSADDSUBJECT(V) 7-12
#EMSGET(V) 7-12
#EMSINIT(V) 7-12
#EMSTEXT(V) 7-12
#EOF 5-16, 5-19
#ERRORTEXT 3-22, 3-27
#EXCEPTION 3-19, 3-24, 3-27, 3-32
#EXTRACT 2-15, 4-2, 5-1, 5-5, 5-16
#EXTRACT(V) 7-3
#EXTRACTV 2-28, 4-11, 5-31, 5-37
#FILEINFO 2-5, 2-34, 4-16, 8-4, 8-8
#FILENAMES 8-8
#FILTER 3-22, 3-25, 3-32
#FRAME 2-3, 2-8, 5-4
#IF 2-4, 2-8
#INFORMAT 2-23
#INLINEECHO 5-7
#INLINEEOF 5-7
#INLINEOUT 5-9, 5-11
#INLINEPREFIX 5-7
#INLINETO 5-9
#INPUT 2-17, 3-22, 3-32
#INPUT(V) 7-5
#INPUTEOF 2-27, 3-28, 6-2, 6-3
#INPUTV 2-5, 2-27, 2-30, 3-28, 6-4, 6-5
#INTERPRETJULIANDAYNO 2-23
#INTERPRETTIMESTAMP 2-23
#JULIANTIMESTAMP 2-18, 2-23
#LINECOUNT 2-11, 2-15
#LINEFINDV 2-11, 5-19
#LOGOFF 2-30
#LOOP 2-3, 2-15, 3-8
#MATCH 2-5, 3-22, 6-2
#MORE 3-5, 3-8, 3-16
#NEWPROCESS 5-2, 5-14, 6-1
#NEXTFILENAME 2-5
#OUTFORMAT 2-23
#OUTPUT 2-7, 2-12, 2-15
#OUTPUTV 2-15
#PREFIX 2-29
#PROCESSEXISTS 6-2

Built-in functions (continued)
#PROCESSINFO 8-3
#PROCESSORSTATUS 8-3
#PROCESSORTYPE 8-3
#PROMPT 2-29
#PURGE 2-5, 4-16
#PUSH 2-3, 2-8
#RAISE 3-29, 3-32
#REPLYPREFIX 6-5, 6-8
#REPLYV 6-5, 7-5
#REQUESTER 2-17, 4-2, 5-1, 5-5, 5-21, 5-25, 5-37, 6-2, 7-3, 7-15
#RESET 3-17, 3-21, 3-27, 3-32
#RESET FRAMES 2-8
#REST 3-5, 3-16
#RESULT 3-8, 3-15
#RETURN 3-17
#ROUTINENAME 2-7, 3-9, 3-16
#SERVER 5-1, 5-31, 6-5, 7-3
#SET 2-3, 2-16
#SETMANY 2-23, 3-27
#SHIFTSTRING 2-5, 3-16, 4-16, 6-2
#SSGET(V) 7-5
#SSINIT 7-5
#SSNULL 7-5
#SSPUT(V) 7-5, 7-13
#SYSTEM 2-29
#SYSTEMNUMBER 3-16
#TACLSECURITY 5-35
#TIMESTAMP 2-18, 2-22
#TOSVERSION 8-3
#UNFRAME 2-8, 5-4
#USERNAME 2-28
#WAIT 4-10, 5-16
Built-in functions, description or example of
Built-in functions:'#OUTPUTV 2-30
Built-in variables
#BREAKMODE 2-28, 3-27
#EXIT 6-3
#INTERACTIVE 6-8
#TRACE 2-32

C

Caller example 3-16
Call_getargs example 3-11
Character address 2-11
CHARACTERS argument alternative 3-3, 3-6
Checkfiles example 2-4
Ckup example 8-1
CLOSEPAREN argument alternative 3-3
CMON process 5-25
COMMA argument alternative 3-3, 3-12
Commands
    for debugging TACL programs 2-32
  ASSIGN 5-3
  FILETOVAR 2-16
  for data manipulation 2-16
  for global editing 2-13
  INLPREFIX 5-7
  PARAM 5-3
  RUN 5-2, 6-1
  SETPROMPT 2-29
  SINK 2-30
  SYSTEM 2-29
  TIME 8-13
  VCHANGE 2-14
  VDELETE 5-19
  VFIND 2-14
  VINSERT 2-13
  VLIST 2-14
  _COMPAREV 2-5
  _CONTIME_TO_TEXT_DATE 8-3
  _MONTH 2-19
Command_processor example 3-21
Completion code 5-33, A-3
Completion information, processing 5-32/34, A-3
Completion variable 5-32, A-3
CONTIME, definition 2-19
Conventions, style 1-2
Copier example 2-2
Copy example 4-10
Creating variables

D

Data types 2-17
  determining 3-5
Date, textual 2-17
Dayofweek example 2-22
Debugging
  sample session 2-33
  _DEBUGGER 2-32 / 35
Defaultvars example 2-8
DEFINE 5-4
Define Process (DP) facility 5-31
Display example 2-25
Displayinfo example 2-27
Distributed System Management
  See DSM
DSM and TACL 7-1
Dumplog example 7-10

E

Editing commands 2-13
Editing variables 2-11
Emptyspool example 5-11
EMS
  and TACL 7-1 / 15
  calling EMSDIST 7-12
  flags 7-13
  generating an event 7-13
  log, displaying contents of 7-10
Enquiry example 5-35
ENQUIRY facility 5-35
Escape sequences for terminals, sending 2-25
Event Management Service
  See EMS
Example
  call_getargs 3-11
  getargs 3-11
Examples
  and #INFORMAT 1-1
  and #PMSEARCHLIST 1-1
  argrec 3-17
  bubble 2-3

Examples (continued)
caller 3-16
checkfiles 2-4
ckup 8-1
command_processor 3-21
copier 2-2
copy 4-10
dayofweek 2-22
defaultvars 2-8
display 2-25
displayinfo 2-27
dumplog 7-10
emptyspool 5-11
enquiry 5-35
fcomp 4-12
fup2 5-18
fupin 5-17
getdates 2-23
inline_fup 5-7
inline_fup_log 5-10
inline_fup_log2 5-11
lock 2-28
menu 2-30
monitoring a system 8-1
nowaited_read 4-4
nowaited_write 4-9
Pathway 6-6
purgefiles 3-22
report_shell 3-15
restricted_caller 3-30
restricted_cmd_processor 3-24
restrictive_command_shell 3-26
running 1-1
runsrv 6-3
same_ssid 7-9
script 5-8
send 5-22
serv 5-30
show_spooler_jobs 5-19
sqlcomp 5-33
strio 5-25
system management 8-1

Examples (continued)
TACL as a Pathway server 6-6
taclevent 7-13
tacllist 4-16
tedsave 2-33
volname 2-12
volnames 2-15
waited_read 4-3
waited_write 4-7
Exception handlers
description of
elements of 3-19
keep type 3-24
release type 3-20
types 3-19
Exception handlers, description of 3-18
Exception, definition of 3-18
EXCLUSION option for #REQUESTER built-in function 4-2

F

Fcomp example 4-12
FILENAME argument alternative 3-3
Files
opening 4-1
reading from 4-2/5
writing to 4-6/9
FILETOVAR command 2-16
Function keys, defining 2-24
Functions, built-in
See Built-in functions
Fup2 example 5-18
Fupin example 5-17

G

Getargs example 3-11
Getdates example 2-23

H

HOLD option for #OUTPUT built-in function 4-16
HOLD option for OUTPUT built-in function

I

IN file 5-1, 5-3
  setting to $RECEIVE 6-3
INLINE facility
  description of 5-1, 5-6/13
  handling _BREAK exception 5-13
  INLINE run option 5-6
  limitations 5-13
  stopping processes 5-13
Inline_fup example 5-7
Inline_fup_log example 5-10
Inline_fup_log2 example 5-11
INLPREFIX command 5-7
INV option
  limitations 5-20
INV option (RUN and NEWPROCESS), NEWPROCESS)
INV variable 5-1

J

Julian day number 2-17
Julian timestamp 2-17

K

KEYWORD argument alternative 3-3

L

Local civil time (LCT) 2-17
Local daylight time (LDT) 2-17
Local standard time (LST) 2-17
Local timestamp 2-17
Lock example 2-28

M

Macros,overview 2-1
Menu example 2-30

N

Nesting code 2-7
Nowaited_read example 4-4
Nowaited_write example 4-9
Numeric date 2-17

O

OPENPAREN argument alternative 3-3
OTHERWISE alternative for ARGUMENT built-in function
OTHERWISE argument alternative 3-6
OUT file 5-1, 5-3
  reading from 5-35
  saving 2-8
  setting to $RECEIVE 6-4
OUTV option
  limitations 5-20
OUTV option (RUN and NEWPROCESS), NEWPROCESS)
OUTV run option 2-15

P

PARAM command 5-3
Pathway example 6-6
Process handles A-5
Processes
  initiating and communicating with 5-1
  initiating and communicating with' 5-37
  stopping 5-20
PROCESSNAME argument alternative 3-3
Programs
  debugging 2-32
  defining structure of 2-2
  exiting 3-17
  style conventions of 1-2
Prompts, setting 2-29
Purgefiles example 3-22

R

Reading from a file
  nowaited 4-4
  waited 4-2
Recursion, in macros and routines 2-7
Report_shell example 3-15
Restricted_caller example 3-30
Restricted_cmd_processor example 3-24
Restrictive_command_shell example 3-26
Results, returning 3-15

Routines
  exception handlers
    combining types 3-30
    keep type 3-24
    release type 3-20
    “keep” type 3-19
    “release” type 3-19
  exception handlers for 3-18, 3-32
  exiting 3-17
  overview 2-1
RUN command 5-2, 6-1
  INV option 5-14
  OUTV option 5-14
Run options
  INLINE 5-6
  STATUS 5-16
Runsrv example 6-3

S

Same_ssid example 7-9
Script example 5-8
Send example 5-22
Serv example 5-30
Server programs
  testing 5-25
Server, TACL as a 6-1 / 12
SETPROMPT Command 2-29
Show_spooler_jobs example 5-19
SINK command 2-30
SLASH argument alternative 3-3
SPI
  and D-series software A-5
  and TACL 7-1 / 12
  buffers 7-5
  definition files 7-4
  messages 7-3
  subsystem ID 7-5
  token code 7-8
  token data types 7-6, A-5
  token map 7-8
Sqlcomp example 5-33

STATUS option (RUN and NEWPROCESS)
STATUS option (RUN) 5-16
STATUS variable 5-20
STRING argument alternative 3-3
String manipulation, built-in functions for 2-10
Strio example 5-25
STRUCT variable
  description of 2-9
  specifying escape sequences for terminals 2-25
  using with D-series software A-5
STRUCT variables
  using with SPI 7-5
Structures
  accessing structured data 5-25
  examples 5-25
SUBSYSTEM argument alternative 3-3
Subsystem ID 7-5
Subsystem Programmatic Interface
  See SPI
Syntax summary
  built-in functions and variables 9-6/14
  STRUCT declarations 9-15
  #DELTA commands 9-17
  #SET built-in variable 9-16/17
  :UTILS:TACL commands and functions 9-1/6
SYSTEM command 2-29

T

TACL
  D-series features A-1
  programs, nesting 2-7
  running as a server
TACL Error
  example 3-26
TACL, running as a server 6-1
TACLCSTM file 6-5
Taclevent example 7-13
Tacllist example 4-16
Tedsave example 2-33
TEMPLATE argument alternative 3-24

Templates
example 3-22
Terminals
sending escape sequences to 2-25
TIME command 8-13
Timestamps 2-17
TOKEN argument alternative 3-3
Token code 7-3
Token code (SPI) 7-8
Token map (SPI) 7-8

U

USER argument alternative 3-3

V

Variable
levels, using 2-3
variable
INV 5-1
OUTV 5-1
:_COMPLETION 5-32
VARIABLE argument alternative 3-8
Variables
creating
modifying 2-9
types 2-1
Variables, editing 2-11
VCHANGE command 2-14
VDELETE command 5-19
VFIND command 2-14
VINSERT command 2-13
VLIST command 2-14
Volname example 2-12
Volnames example 2-15

W

WAIT option for #REQUESTER built-in function 4-2
Waited_read example 4-3
Waited_write example 4-7
WORD argument alternative 3-8

WORDLIST argument alternative 3-5
Writing to a file
  nowaited 4-8
  waited 4-6

Z

ZSPIDEF files 7-4, 7-5
ZSPISEGF files 7-4

Special characters

#built-in funcion
  See Built-in functions
#built-in variable
  See Built-in variables
$CMON 5-25, A-2
$RECEIVE 5-1, 5-5, 7-3
  and TACL as a server 6-1
  using for process communication 5-21/29
%0%, using to call a macro recursively 2-7
:_COMPLETION' variable; 5-32
_COMPAREV command 2-5
_COMPLETION^PROCDEATH variable A-3
_CONTIME_TO_TEXT_DATE command 8-3
_DEBUGGER commands 2-32
_MONTH function 2-19
_PROMPTER 2-29