Inspect Manual

Abstract

This manual describes Inspect, a command-line tool for debugging TNS C/C++, COBOL, FORTRAN, Pascal, Screen COBOL, and TAL programs and snapshots on HP NonStop™ TNS/R and TNS/E systems.

Product Version

Inspect H01

Supported Release Version Updates (RVUs)

This publication supports J06.03 and all subsequent J-series RVUs, H06.03 and all subsequent H-series RVUs, G01.00 and all subsequent G-series RVUs, and D40.00 and all subsequent D-series RVUs, until otherwise indicated by its replacement publications.

Part Number	Published
429164-006	April 2013

Document History

Part Number	Product Version	Published
118810	Inspect D40	December 1995
429164-002	Inspect H01	July 2005
429164-003	Inspect H01	January 2006
429164-004	Inspect H01	August 2010
429164-005	Inspect H01	February 2012
429164-006	Inspect H01	April 2013

Legal Notices

Inspect Manual

Glossary | Index | Examples | Figures | Tables

1. Introduction (continued)

2. Inspect Concepts

6. High-Level Inspect Commands

6. High-Level Inspect Commands (continued)

6. High-Level Inspect Commands (continued)

6. High-Level Inspect Commands (continued)

6. High-Level Inspect Commands (continued)

6. High-Level Inspect Commands (continued)

6. High-Level Inspect Commands (continued)

6. High-Level Inspect Commands (continued)

6. High-Level Inspect Commands (continued)

6. High-Level Inspect Commands (continued)

8. Using Inspect With C (continued)

9. Using Inspect With C++

9. Using Inspect With C++ (continued)

10. Using Inspect With COBOL and SCREEN COBOL

13. Using Inspect With TAL and pTAL (continued)

B. Syntax Summary

B. Syntax Summary (continued)

B. Syntax Summary (continued)

C. Notes for System Operators

Tables (continued)

What's New in This Manual

Manual Information

Abstract

This manual describes Inspect, a command-line tool for debugging TNS C/C++, COBOL, FORTRAN, Pascal, Screen COBOL, and TAL programs and snapshots on HP NonStop™ TNS/R and TNS/E systems.

Product Version

Inspect H01

Supported Release Version Updates (RVUs)

This publication supports J06.03 and all subsequent J-series RVUs, H06.03 and all subsequent H-series RVUs, G01.00 and all subsequent G-series RVUs, and D40.00 and all subsequent D-series RVUs, until otherwise indicated by its replacement publications.

Part Number	Published
429164-006	April 2013

Document History

Part Number	Product Version	Published
118810	Inspect D40	December 1995
429164-002	Inspect H01	July 2005
429164-003	Inspect H01	January 2006
429164-004	Inspect H01	August 2010
429164-005	Inspect H01	February 2012
429164-006	Inspect H01	April 2013

New and Changed Information

Changes to the 429164-006 manual:

- Added a new paragraph on page 2-5.

Changes to the H06.24/J06.13 manual:

- Added new error messages, 1106, 1107 on page A-50.

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

- Supported release statements have been updated to include J-series RVUs.
- Added a new section, Limitation of the STEP Command on page 6-215.
- Updated the description of IMON and CMON on page C-2.

Changes in the January 2006 revision of the Manual

- Updated an example for the MODIFY command to include the & operator on page 6-151.
- Updated a usage guideline for COBOL programmers about the & operator in the MODIFY command under MODIFY on page 10-13.
- Added a missing error, 123 on page A-21.

Changes to the Original Inspect H01 Manual

- Throughout the manual:
  - Updated the product information and company names that have been changed during the previous RVUs.
  - Added the new system procedures PROCESS_CREATE and PROCESS_LAUNCH with NEWPROCESS and NEWPROCESSNOWAIT procedures. For example, see Starting a Debugging Session on page 4-8, What Inspect Debugs on page 2-4, and Debug Events on page 2-8.
- Added a new Inspect feature Debugging Programs Executed on TNS/E Systems on page 1-5.
- Updated the native linkers for TNS/R and TNS/E systems under Processes on page 2-4.
- Added a new table that lists rules for the debugger selection under The Debugging Attributes of a Process on page 4-4.
- Added a new figure, Figure 4-3 on page 4-6 to illustrate the rules for debugger.
- Added a note to use either Visual Inspect or Native Inspect to debug a Pathway server under Debugging PATHWAY Servers on page 5-10.
- The changes made in Section 6, High-Level Inspect Commands are:
  - Added a new usage consideration for the ADD Program on page 6-13.
  - Updated the description of BACKUP on page 6-20 and BOTH on page 6-43.
  - Usage Considerations for Accelerated Programs on TNS/E Systems on page 6-98 shows safe-point annotations for an OCA process but not for a snapshot of an OCA process.

    - Added the information that the BOTH option does not show TNS/E instructions for an OCA process on a TNS/E system on page 6-97.
    - Updated the usage guideline of INFO OBJECTFILE on page 6-111 and INFO SAVEFILE on page 6-120.
    - Added two examples to show the listing for an accelerated TNS program running on a TNS/R and TNS/E system on page 6-140 and 6-141.
    - The SELECT DEBUGGER DEBUG on page 6-165 selects Native Inspect instead of Debug on TNS/E systems.
    - Added a limitation that CODE or LIB cannot be used to specify a TNS/E native object file on page 6-168.
    - Updated the usage consideration for Accelerated programs on TNS/R and TNS/E systems on page 6-201.
    - Updated the usage consideration for TRACE on page 6-221.
- Section 16 title has been changed from "Using Inspect With Accelerated Programs" to Section 16, Using Inspect With Accelerated Programs on TNS/R Systems. A new figure, Figure 16-1 on page 16-2 has been added to illustrate acceleration by Axcel.
- Added a new subsection Dynamic-Link Libraries (DLLs) on page 17-5.
- Added a new section,Section 18, Using Inspect on a TNS/E System that describes how to use Inspect to debug emulated TNS processes running on a TNS/E system.
- Added new error messages, 406 on page A-48, 450 on page A-49, and 1003 on page A-49 through 1009 on page A-50.
- Added a new error under DMON Errors in Appendix C on page C-6.

About This Manual

This manual describes the Inspect interactive symbolic debugger for TNS/R and TNS/E systems. This manual is intended for system and application programmers.

Related Documentation

For more information about debugging on the TNS/E platform, see:

- Inspect online help (and the description of HELP on page 6-91).
- *TNSVU Manual*
- *Native Inspect Manual* and Native Inspect online help
- Visual Inspect online help

Organization of This Manual

Table i lists and describes the sections and appendixes in this manual.

Table i. Contents of the *Inspect Manual*

Section	Title	Contents
1	Introduction	Contains an overview of Inspect, including a definition of Inspect, its features, command modes, and components.
2	Inspect Concepts	Explains basic and advanced concepts, including what Inspect can debug, execution states, debug events, and language-specific Inspect elements. Also describes Inspect terms such as scope path, code location, and data location.
3	Inspect Command Overview	Introduces the high-level Inspect commands, grouping them by function. Also describes how to enter Inspect commands.
4	Debugging Processes and Save Files	Explains how to use Inspect to debug processes and to examine save files.
5	Debugging PATHWAY Applications	Explains how to use Inspect to debug Pathway applications: requester programs, servers, and user conversion routines.
6	High-Level Inspect Commands	Contains an alphabetic list of all high-level Inspect commands, including descriptions, syntax, default values, and examples.
7	Low-Level Inspect	Describes low-level Inspect, contrasting it with DEBUG and high-level Inspect. Also presents the low-level Inspect commands.

Table i. Contents of the *Inspect Manual*

Section	Title	Contents
8	Using Inspect With C	Provides the language-specific information necessary to debug C programs using Inspect.
9	Using Inspect With C++	Provides the language-specific information necessary to debug C++ programs using Inspect.
10	Using Inspect With COBOL and SCREEN COBOL	Provides the language-specific information necessary to debug COBOL 74, COBOL85, and Screen COBOL programs using Inspect.
11	Using Inspect With FORTRAN	Provides the language-specific information necessary to debug FORTRAN programs using Inspect.
12	Using Inspect With Pascal	Provides the language-specific information necessary to debug Pascal programs using Inspect.
13	Using Inspect With TAL and pTAL	Provides the language-specific information necessary to debug TAL and pTAL programs using Inspect.
14	Using Inspect in an OSS Environment	Provides information necessary to debug in the OSS environment using Inspect.
15	Section 15, Using Inspect on a TNS/R System	Provides information necessary to debug accelerated programs using Inspect on a TNS/R system.
16	Using Inspect With Accelerated Programs on TNS/R Systems	Provides information necessary to debug accelerated TNS programs using Inspect on a TNS/R system.
17	Section 17, Using Inspect With TNS/R Native Programs	Provides information necessary to debug TNS/R native programs using Inspect on a TNS/R system.
18	Using Inspect on a TNS/E System	Describes debugging emulated TNS programs and TNS/R snapshots on a TNS/E system.
Appendix A	Appendix A, Error and Warning Messages	Contains a numeric list of all error and warning messages that Inspect produces.
Appendix B	Appendix B, Syntax Summary	Provides a summary of high-level Inspect commands. It also provides a summary of the language-specific information for each source language.
Appendix C	Appendix C, Notes for System Operators	Discusses topics concerning the installation and maintenance of Inspect.
Glossary	Glossary	Defines Inspect and related terms.

Notation Conventions

Hypertext Links

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

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

General Syntax Notation

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

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

```
MAXATTACH
```

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

file-name

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

```
myfile.c
```

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

pathname

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

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

INT[ERRUPTS]
```

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

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

K [ X | D ] address
```

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

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

```
ALLOWSU { ON | OFF }
```

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

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

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

```
M address [ , new-value ]…
```

```
[ - ] {0|1|2|3|4|5|6|7|8|9}…
```

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

```
"s-char…"
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Change Bar Notation

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

> The message types specified in the REPORT clause are different in the COBOL85 environment and the Common Run-Time Environment (CRE).
>
> The CRE has many new message types and some new message type codes for old message types. In the CRE, the message type SYSTEM includes all messages except LOGICAL-CLOSE and LOGICAL-OPEN.

HP Encourages Your Comments

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

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

1 Introduction

Inspect Features

Inspect is an interactive symbolic debugger that enables you to isolate program bugs quickly. Inspect provides several features that help you shorten the debugging phase of a project:

- Interactive control of program execution
- Concurrent debugging of all parts of an application
- Debugging of PATHWAY applications
- Support for saving and examining a process state
- Source-level and machine-level program access
- Support for many source languages
- Support of optimizing compilers
- Support for programs written in multiple source languages
- Local system customization
- Personal customization
- Code and data breakpoints
- Conditional breakpoints
- Distributed debugging across a network
- Debugging programs executed on TNS/R systems
- Debugging programs executed on TNS/E systems

- Debugging programs in an OSS environment

Interactive Control of Program Execution

Inspect enables you to look at your program while it is running. Without interactive control of program execution, take these actions:

1. Edit your source code, adding statements that write out status messages.
2. Recompile your source program.
3. Execute the resultant object program.
4. Examine the status messages.
5. Return to Step 1.

Interactive execution control enables you to bypass this time-consuming procedure. It enables you to stop and resume execution of your program selectively. Using this “stop and go” technique, you can quickly find out how your program is malfunctioning.

Using an interactive debugger is more efficient than using the primitive cycle. For example, communicating by telephone is more efficient than by telegram.

Concurrent Debugging of All Parts of an Application

Large applications tend to consist of several programs working together, passing status messages and data back and forth to each other. Inspect enables you to debug such multiprogram applications, because it can debug more than one program at a time. Although Inspect permits concurrent debugging of several programs, it tracks each program separately. This separate treatment enables to debug a multiprogram application as easily as a single-program application.

Debugging of PATHWAY Applications

PATHWAY uses the requester/server approach to application design. The requester provides the terminal management functions, and the server provides the database management functions. Inspect, with the PATHWAY Terminal Control Process (TCP), enables you to debug requesters, servers, or complete PATHWAY applications.

Saving and Examining a Process State

Inspect allows the state of processes that are being debugged to be saved to a save file. In addition, if you enable the SAVEABEND attribute of a process, DMON will create a file containing an image of that process if it abends. You can use Inspect to examine this file, enabling you to view a process in a normally inaccessible state—at abnormal termination. In addition, Inspect allows the state of processes that are being debugged to be saved to a save file.

Source-Level and Machine-Level Program Access

When you debug a program written in a high-level language, the debugger should provide source-level access to the program. Machine registers, absolute addresses, and internal storage schemes generally are not a concern. Inspect provides this type of source-level, or symbolic, access. The symbolic access provided by Inspect lets you:

- Refer to a location by the high-level name you've given it in your source code.
- Refer to a source location by its line number in a source file.
- Evaluate expressions using the operators and operator precedence of your source language.
- Display source code.
- Display and modify the current data segment ID.
- Control program execution on a statement-by-statement basis.

Symbolic debugging eases the task of debugging and decreases your debugging time, but sometimes you need to have machine-level access to your program. The machine-level access provided by Inspect lets you:

- Refer to an item by its absolute or relative address.
- Display and modify machine registers.
- Display machine code.
- Control program execution on an instruction-by-instruction basis (except for accelerated programs running on a TNS/R machine). For more information, see Section 16, Using Inspect With Accelerated Programs on TNS/R Systems.

Support for Many Source Languages

Inspect enables you to debug applications written in any of these source languages:

C
C++
COBOL (COBOL 74 or COBOL85)
FORTRAN
Pascal
pTAL
SCREEN COBOL
TAL

High-level Inspect supports the same set of commands for all the languages, with minor variances for language-dependent extensions. For example, to ask Inspect for the current value of a data item name `rate`, enter the high-level command `DISPLAY rate`, regardless of the program's source language.

Although the functions and names of high-level commands are uniform across all supported languages, the parameters to several of the commands are language-dependent. Therefore, you have the flexibility needed to offer you symbolic access to your program. When debugging a COBOL program, use this COBOL syntax to refer to a data item:

```
DISPLAY first-name OF manager OF department (dept-num)
```

However, when debugging a TAL program, use this TAL syntax:

```
DISPLAY department[dept^num].manager.first^name
```

The Inspect command is the same (DISPLAY), but the way you refer to the data item is different.

Support of Optimizing Compilers

Inspect recognizes optimization performed across statements, but it issues a message when optimization has:

- Removed the code for a statement
- Altered the code so that Inspect might report incorrect results

For details about optimization, see the respective language reference manual. For more information about Inspect and optimization, see Debug Events on page 2-8 and Code Locations on page 2-16.

Support for Programs with Multilanguage Source

Using the Binder (TNS) or nld (TNS/R) utility, you can create programs whose source code is written in more than one language. Inspect enables you to debug these multi-source programs. Inspect also automatically selects the appropriate source language to use to interpret high-level Inspect commands.

Local System Customization

When an Inspect session begins, Inspect looks for an EDIT file named INSPLOCL in the volume and subvolume containing the Inspect program file. If this INSPLOCL file exists, Inspect reads and executes the Inspect commands it contains. Consequently, your system manager can customize the Inspect environment for all Inspect sessions that run on your system.

Personal Customization

After reading the INSPLOCL file, Inspect looks for an EDIT file named INSPCSTM in your logon volume and subvolume. If this file exists, Inspect reads and executes the Inspect commands it contains. Consequently, you can customize your Inspect environment by creating an INSPCSTM file that includes Inspect commands to configure the environment any way you want. Note that the commands in the INSPCSTM file override those in the INSPLOCL file because Inspect processes the INSPCSTM file after the INSPLOCL file. When debugging PATHWAY requestor programs, your logon volume and subvolume may differ from the owner of the process ID of the TCP’s volume and subvolume.

Code and Data Breakpoints

Inspect lets you set breakpoints at both code and data locations within your program. In addition, Inspect lets you select what type of access (write or read/write or change) triggers a data breakpoint. Note that write access is not available on TNS/R systems.

Conditional Breakpoints

Inspect lets you limit a breakpoint so that it suspends program execution only if certain conditions exist. This enables you to avoid repeated execution interruptions before arriving at the location of a bug.

Distributed Debugging Across a Network

Inspect enables you to debug applications whose components are distributed across an EXPAND network. The distributed components can include:

Processes or programs
Source files
Program (object) files

Debugging Programs Executed on TNS/R Systems

Inspect supports the debugging of TNS, accelerated TNS, and native TNS/R programs on TNS/R systems.

Debugging Programs Executed on TNS/E Systems

Inspect supports the debugging of emulated TNS programs on TNS/E systems, in addition to snapshots of TNS/R native processes. Emulated TNS programs are TNS programs that either have been accelerated beforehand with the TNS Object Code Accelerator (OCA) or run in TNS interpreted mode.

Debugging Programs in the OSS Environment

Inspect supports the debugging of programs in the Open System Services (OSS) environment. Processes that run in the OSS environment use the OSS application program interface. Interactive users of the OSS environment use the OSS shell for the command interpreter. Debugging support in the OSS environment includes OSS file descriptors, signals, and program IDs (PIDs).

Although there are no TNS OSS processes on TNS/E systems, you can use Inspect to debug a TNS OSS snapshot on a TNS/E system.

Inspect Command Modes

Inspect provides two command modes: high-level mode and low-level mode. High-level mode provides source-level access to your program, while low-level mode provides machine-level access.

Each mode has its own set of commands, but they offer many of the same functions. Besides display, modify, break, and trace capabilities, both modes provide convenience commands such as HELP, FC, LOG, and OBEY.

Note. You can debug SCREEN COBOL programs in high-level mode only.

High-Level Mode

To take full advantage of high-level Inspect, you should ensure that your program file includes symbol information. Consequently, you must include the SYMBOLS compiler directive in the source code or on the command line when you compile the source code.

High-level Inspect provides features that let you:

- Refer to code and data locations using source identifiers.
- Modify the value or values of a data item.
- Display the value or values of a data item, with extensive control of the display format.
- Step through your program in source or machine oriented increments.
- Provide aliases for command strings.
- Suspend program execution and automatically perform a specified action.
- Save an image of a process for later examination.

Low-Level Mode

Low-level Inspect resembles Debug. To use either effectively, you must understand the architecture of HP NonStop computer systems.

Your program file does not need to include symbol information to use low-level (or high-level) Inspect. If it does have this information, you can use the high-level INFO command instead of compiler listings or binder maps to discover the address, addressing mode, and size of any symbol in your program. However, if your object code does not contain symbols, you can point Inspect at a corresponding version of the object.

Low-level Inspect offers these special features that are not available in Debug:

- Recognition of source-language names of code blocks.
- Stepping through your program by machine instructions (on TNS/R systems, this can be done with non-accelerated programs only).

Automatic Command Mode Selection

When the execution of your program is suspended, Inspect automatically selects its command mode, depending on the availability of symbol information.

If the scope containing the current execution location includes symbol information, Inspect automatically enters high-level mode; otherwise, it enters low-level mode. Once Inspect has made its selection, you can switch to the other command mode using the LOW command in high-level Inspect or HIGH command in low-level Inspect.

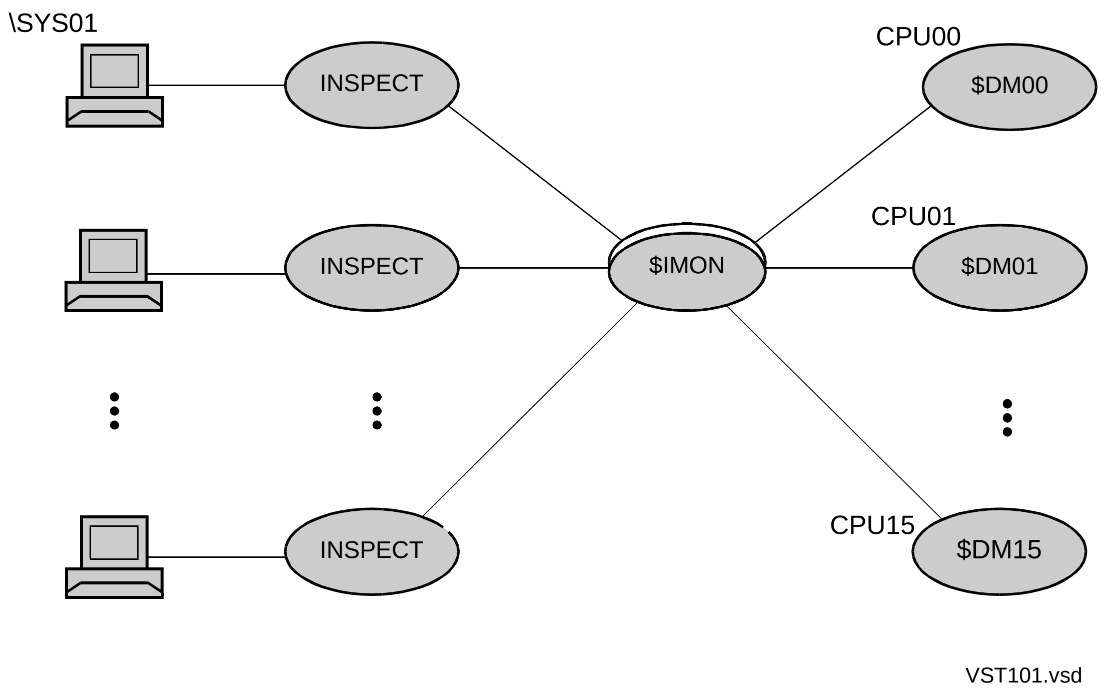
Inspect Components

The Inspect interactive symbolic debugger consists of three groups of processes: Inspect processes, the IMON process pair, and DMON processes. These component groups perform these functions:

- Inspect processes provide the terminal interface to Inspect. There is one Inspect process for each terminal in use for debugging.
- The IMON process pair monitors the operation of Inspect for an entire system. There is one IMON process pair for each system.
- DMON processes provide the execution control facilities of Inspect. There is one DMON process for each CPU within a system.

Figure 1-1 presents a conceptual overview of these component processes and their interaction.

Figure 1-1. Inspect Components



The Inspect Process

An Inspect process provides the terminal interface through which you interact with the Inspect debugger. The other functions of an Inspect process include:

- Retrieving source code from source files.
- Retrieving machine code from program files.
- Retrieving symbol information from program files.
- Communicating with DMON, making execution control and program data requests.

IMON starts an Inspect process on the home terminal of a process for which a debugging event occurs or on which a debug request is issued, or if you enter the command interpreter Inspect command. Note that IMON does not start Inspect, if you enter the command interpreter Inspect command, for example RUN INSPECT and INSPECT.

The IMON Process Pair

The IMON process runs as a fault-tolerant process pair named $IMON. Your system manager starts IMON at the same time as the operating system, and determines what processors run IMON. The functions of IMON include:

- Starting a DMON process on each processor in your system
- Monitoring the status of the DMON processes, ensuring that a DMON is always running in each CPU
- Starting Inspect processes

The DMON Process

A DMON process runs on each processor in a system that has IMON running. The name of the DMON process running on processor number *nn* is $DM*nn*. A DMON process provides debugging functions for all programs running in its processor; these functions include:

- Setting or clearing data and code breakpoints
- Retrieving values from or storing values in the program's data space
- Informing Inspect when a breakpoint has been reached

Creating an image of a program and storing it in a save file for later examination

Note. The TCP, not DMON, provides these execution control functions for PATHWAY requester programs.

Inspect, IMON, and DMON Swap and Extended Swap Files

The Inspect, IMON, and DMON swap files do not need to be located on the $SYSTEM volume. IMON and DMON swap files are on the swap file volume specified when the $IMON process is started. Inspect processes that are started by IMON have the same swap file volume as the first process that is debugged; for example:

```
> ALTER DEFINE =_DEFAULTS, SWAP $DISK4
> IMON /NAME $IMON, SWAP $DISK4/ -- IMON and DMON swap files are on $DISK4
       .
       .
       .
> RUND -- INSPECT’s extended swap files are also on $DISK4
```

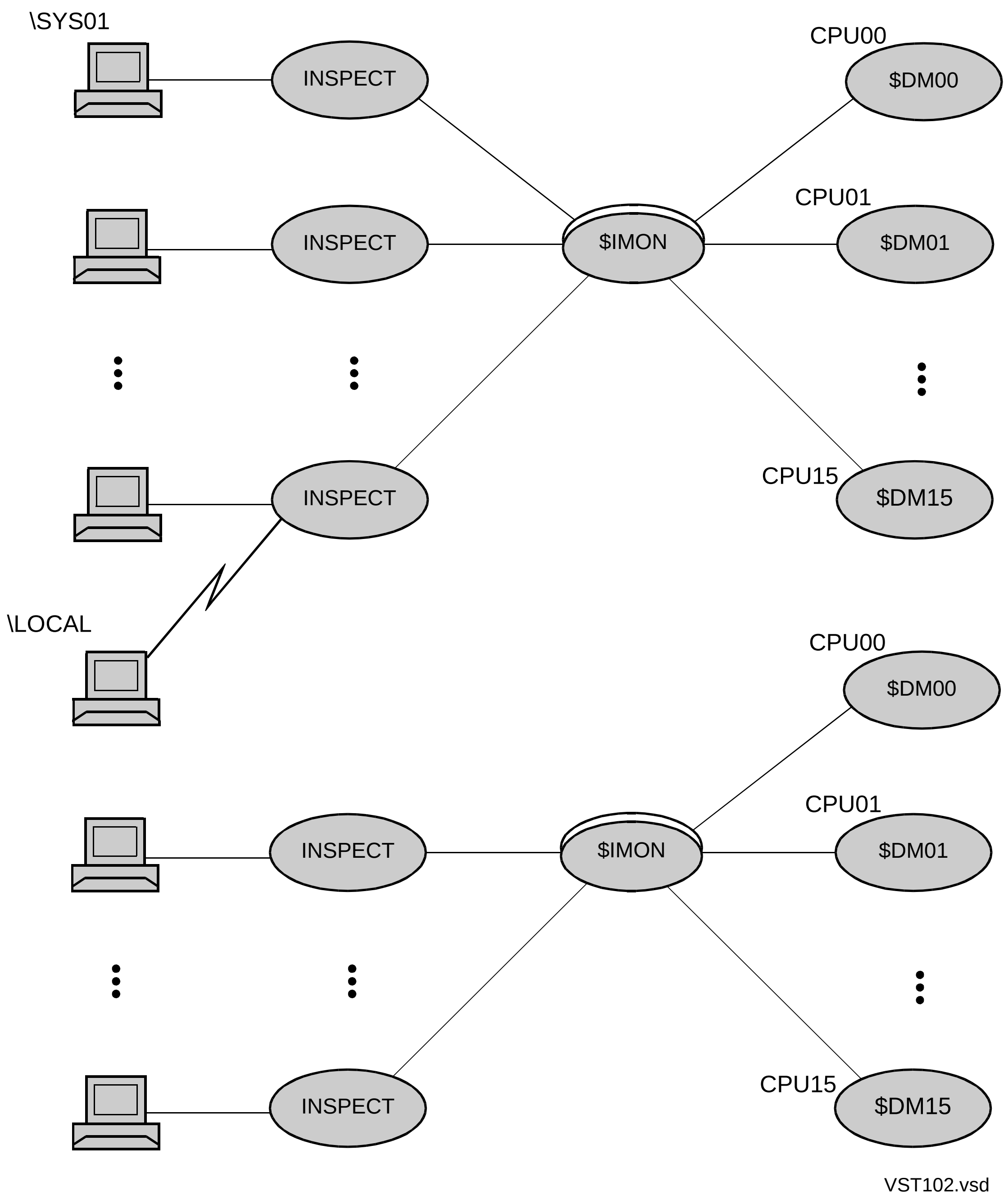
For SCREEN COBOL programs, the swap file volume of the TCP is used, which is the volume specified in the TCP configuration parameter, “Guardian Swap.” To override, start Inspect from the command interpreter Inspect command. Note that IMON does not start Inspect if you enter the command interpreter Inspect command.

```
> INSPECT / SWAP $DISK4, EXTSWAP $DISK4/ -- INSPECT's swap
and extended swap files are on $DISK4
```

Remote Debugging

Inspect enables you to debug a program running on another system in an EXPAND network. When debugging such a program, the Inspect process runs on the remote system. If you want to debug a process on a remote node, the process must have been started remotely or you will receive a security error from the operating system on the remote node. Figure 1-2 shows the configuration when debugging across a network.

Figure 1-2. Inspect Across a Network



2 Inspect Concepts

Inspect Sessions

An Inspect session is the time period during which you debug interactively using the same Inspect process. The session begins when IMON first creates an Inspect process on your terminal, and continues until you terminate that process or exit Inspect.

Starting an Inspect Session

There are four ways to start an Inspect session:

- Run Inspect directly.
  - From the TACL prompt, enter:

    ```
    >RUN INSPECT
    ```

    or:

    ```
    >INSPECT
    ```
  - From the OSS shell, enter:

    ```
    gtacl -p inspect
    ```
- Run your program with the debug option.
  - From the TACL prompt, enter:

    ```
    >RUND program
    ```
  - From the OSS shell, enter:

    ```
    run -debug -inspect=on program
    ```
- Issue a debug request on a running program.
- A debug event occurs in a program that has your terminal specified as its home terminal or debugging terminal. For more information, see Debug Events on page 2-8.

The first of these events always starts a new Inspect session. The second, third and forth start a new Inspect session if your terminal does not already have an Inspect session.

The first two options have been enhanced to support the OSS shell. In all cases, Inspect will be started as a Guardian process.

To avoid incompatibilities between Guardian and the OSS terminal I/O when debugging an OSS process, change Inspect's command terminal to an existing Guardian terminal using the TERM command as soon as Inspect comes up.

At the start of an Inspect session, the Inspect process displays a banner similar to this:

```
INSPECT - Symbolic Debugger - T9673D40 - (29JUL03)   System \sysname
Copyright Tandem Computers Incorporated 1983, 1985-1993
--
```

This banner signifies the start of the session (\sysname is the system on which the Inspect process is running).

After printing the banner, the Inspect process looks for an EDIT file named INSPLOCL in the volume and subvolume containing the Inspect program file. If this INSPLOCL file exists, Inspect reads and executes the Inspect commands it contains.

Consequently, your system manager can customize the Inspect environment for all Inspect sessions that run on your system.

After reading the INSPLOCL file, the Inspect process looks for an EDIT file named INSPCSTM in the logon volume and subvolume of the creator of the process being debugged. However, for PATHWAY programs, Inspect uses the default volume and subvolume of the individual who started PATHMON.

If the INSPCSTM file exists, Inspect reads and executes the Inspect commands it contains. Consequently, you can customize your Inspect environment by creating an INSPCSTM file that includes Inspect commands to configure the environment the way you like it. Note that the commands in the INSPCSTM file override those in the INSPLOCL file because Inspect processes the INSPCSTM file after the INSPLOCL file.

Prompting for Commands

Whenever Inspect expects you to enter a command, it prints the Inspect prompt. By default, the prompt has this form:

```
-PRGOBJ-
```

The dashes enclosing the program name (PRGOBJ) signify that Inspect is in high-level mode; underscores enclosing the program name, as in _PRGOBJ_, denote low-level mode.

The name between the dashes is the name of the program that you are currently debugging. If you are not currently debugging any program, Inspect prompts you with two dashes or two underscores, depending upon the current mode.

The SET PROMPT command enables you to customize the Inspect prompt to show other types of information as well. For more information, see SET PROMPT on page 6-178.

Reporting Events

When a debug event occurs, Inspect displays a status message that provides information regarding the event. By default, the status message has this form:

```
INSPECT  BREAKPOINT 1: #MAIN
175,05,00066  TALOBJ  #MAIN.#1862(TALSRC)
```

The message begins with the text “INSPECT” to indicate that Inspect is generating the message. Following this text is a description of the type of event that occurred. In the example, the debug event is a break event caused by breakpoint number 1. The definition of the breakpoint (#MAIN) follows the breakpoint number.

The second line of the message indicates where the event occurred. Inspect displays the PID and name of the program in which the event occurred. It then shows the code location where the event occurred: line number 1862 within the scope unit MAIN, whose source file is TALSRC.

The SET STATUS command enables you to customize Inspect event reporting. For more information, see SET STATUS ACTION on page 6-186 and SET STATUS LINE25 and SET STATUS SCROLL on page 6-187.

Ending an Inspect Session

An Inspect session can be stopped in several ways:

- You exit, either by using the EXIT command or the RESUME * EXIT command
- Your last program has completed
- Your program is stopped by someone else

When you use EXIT to end an Inspect session, Inspect will prompt you for confirmation. If you have not removed all breakpoints from the programs you have been debugging before attempting to exit Inspect, Inspect will mention that all breakpoints have not been removed and prompt you for confirmation.

What Inspect Debugs

You can debug or examine any of these program types with Inspect:

- Processes
- Save files
- PATHWAY servers
- PATHWAY requester programs

Inspect groups these four types under the common term “program.”

Note. Inspect’s usage of the term program differs slightly from the general usage. Inspect defines program as a file that contains a series of instructions. In this manual, such a file is called a program file, and the term program is used to mean “a process, a save file, a PATHWAY server, or a PATHWAY requester program.”

Processes

Processes are running machine-code programs. More specifically, a process is the unique executing entity that is created when:

- Someone runs object code from a program file by entering an explicit or implicit TACL RUN command.
- An existing process runs object code from a program file by calling the NEWPROCESS, NEWPROCESSNOWAIT, PROCESS_CREATE_, or PROCESS_LAUNCH_ system procedure.

Inspect enables you to debug processes started from program files whose object code was compiled from C, COBOL, COBOL85, FORTRAN, Pascal, or TAL source code.

With Inspect, you can also debug processes started from program files whose object code was produced from one or more object files (compiled from any of these languages) either by the BINDER utility or by a native linker such as nld (on TNS/R systems), ld (for Position-Independent Code on TNS/R systems), or eld (on TNS/E systems).

Inspect allocates heap during debugging a process (that is, setting a breakpoint, resuming, displaying a variable and for various other Inspect debugging commands). The limit for this heap for Inspect is 127.5 MB. If this limit exceeds, Inspect terminates.

Inspect also supports the debugging of TNS processes, in addition to emulated TNS processes on TNS/R and TNS/E systems. An emulated TNS process is a TNS process whose object file has been processed by the Axcel accelerator (on a TNS/R system) or by OCA (on a TNS/E system) to optimize the process for the native system. For more information about using Inspect on a TNS/E system, see Section 18, Using Inspect on a TNS/E System.

Many of the high-level Inspect commands permit you to refer to items using the names or identifiers you gave them in your source code. To take full advantage of these commands, you should ensure that your program file includes symbol information. Consequently, you must include the SYMBOLS compiler directive in the source code or on the command line when you compile the source code.

Save Files

Save files are “snapshots” or “images” of a process or PATHWAY server stored on disk. If a process or PATHWAY server terminates abnormally (abends) and its SAVEABEND attribute is set, DMON creates a save file just before termination. You can direct DMON to create a save file using the Inspect SAVE command.

A save file contains information regarding the state of the process or PATHWAY server at the moment the image was created. This information includes:

- The user data space (including extended data segments)
- The values of machine registers
- The names and status of any files opened by the process or PATHWAY server
- Information about the programs execution environment.

Save files contain additional information about the TNS/R execution environment. This includes an indication of whether the program has been accelerated and the current instruction set that is being executed. For accelerated and TNS/R native programs, it also stores TNS/R machine registers.

Inspect pairs a save file with the program file that created the saved process; it therefore enables you to examine a “frozen” process. Such examination makes the task of finding and eliminating fatal bugs (those that cause an ABEND) much easier.

Note. Save files include an indication of whether the program has been accelerated and the current instruction set that is being executed. For accelerated programs, it also stores TNS/R machine registers.

Save files are not upwardly compatible. You will receive this error message if you attempt to add a save file newer than the current version of Inspect to an older version of Inspect.

```
** Inspect error 90 ** Incompatible Save File version
```

PATHWAY Servers

A PATHWAY server is the half of a PATHWAY application that manages database files. Some of the tasks handled by a PATHWAY server include:

- Updating the database files (additions, deletions, corrections)
- Retrieving information from the database files
- Communicating with TCPs

PATHWAY servers are processes, so they share most of the traits of processes; however, because the PATHWAY system oversees the creation and operation of servers, servers differ from processes in some important ways, including how you start them. You start a PATHWAY server by entering the START SERVER command from PATHCOM.

PATHWAY Requester Programs

A PATHWAY requester program is the half of a PATHWAY application that manages transaction data at the terminal. Some of the tasks handled by a PATHWAY requester program include:

- Displaying data-entry forms on terminals associated with the application
- Accepting data entered at the terminals
- Checking input for errors
- Passing data requests and updates along to a PATHWAY server for processing

PATHWAY requester programs are written in SCREEN COBOL and compiled into an intermediary form of object code called pseudocode, which is executed by a Terminal Control Process (TCP). Compilation to pseudocode distinguishes PATHWAY requester programs from processes and PATHWAY servers because they are compiled directly into machine code, which is then executed by a processor.

PATHWAY requester programs can be started by entering one of these PATHCOM commands:

- START TERM starts the PATHWAY requester program associated with a given terminal by an earlier SET TERM command.
- RUN PROGRAM starts a given PATHWAY requester program at the PATHCOM command terminal.
- Inspect enables you to debug PATHWAY requester programs compiled with the SCREEN COBOL directive SYMBOLS.

Debugging Multiple Programs

Inspect enables you to debug more than one program in a single Inspect session; in fact, you can debug several programs concurrently. When you debug multiple programs, Inspect retains a list of the programs that you are debugging. This list is called the program list.

Although you can debug multiple programs concurrently, most Inspect commands affect only one program: the current program. Whenever a debug event occurs in one of the programs you're debugging, Inspect automatically makes it the current program. In addition, you can choose the current program using the SELECT PROGRAM command.

Execution States of a Program

Inspect separates the execution of a program into four distinct phases, called execution states. These states describe the current activity of a program, and determine the availability and significance of certain Inspect commands. Because save files are not executable, the execution state of a save file cannot change. A save file merely records the execution state of the process that it is an image of.

All other program types can assume one of three possible Inspect execution states:

- Run State—the program is executing.
- Hold State—the program is temporarily suspended, but can resume execution again.
- Stop State—the program has completed execution, whether normally or abnormally.
- Gone State—the program has stopped executing, but has not been cleared.

Several Inspect commands produce relevant information only when a program is in the hold or stop state, while other commands are valid only when a program is in the hold state.

The Run State

A program is in the run state as long as it is executing. A program enters the run state when it is first created (and it is not configured to enter the hold state immediately) or when you enter a RESUME or STEP command to release it from the hold state.

A program leaves the run state when it completes execution, whether normally or abnormally, or when a debug event occurs. In the former case, the program enters the stop state. In the latter case, the program enters the hold or stop state, depending on the nature of the debug event.

The Hold State

A program is in the hold state while its execution is suspended. A program enters the hold state as the result of a debug event. Many Inspect commands are valid only when the program is in the hold state. For example, you can use MODIFY to change data values only when execution is suspended.

A program leaves the hold state (and re-enters the run state) when you enter a RESUME or STEP command.

The Stop State

A program is in the stop state just after it completes execution, whether normally (STOP) or abnormally (ABEND). A program enters the stop state when it calls the STOP or ABEND system procedure, when another program stops it by calling the STOP system procedure, or when you stop it using the TACL STOP command.

Normally a program is removed (that is, its code and data areas are freed for other use) after it enters the stop state; however, a program is also in the stop state if:

- The program was suspended by a breakpoint set on STOP/ABEND.
- You examine a save file that was created from a program in the stop state.

Once a program is in the stop state, it cannot enter the run or hold state. Furthermore, you cannot set breakpoints in it, modify its data, or resume its execution.

The Gone State

A program is in the gone state after you have stopped program execution and a message has been sent to DMON. Processes only exist in the gone state for as long as it takes DMON to compete the stopping process. Your program is not cleared from the list until DMON returns a message. It is possible to catch a program you have stopped if the program is still in the gone state.

Debug Events

A debug event is an action that causes Inspect to suspend the execution of a program. Because save files are not executable, debug events do not apply to them.

All these actions cause a debug event to occur:

- You enter the DEBUG or RUND command from TACL.
- You enter the START TERM command from PATHCOM for a terminal that you've configured with a SET TERM INSPECT ON command.
- You enter an Inspect command from PATHCOM.
- You enter a HOLD or ADD PROGRAM command from Inspect.
- Program execution (or termination) activates a breakpoint, and the breakpoint triggers a break event.
- A TACL routine or macro calls the built-in TACL function #DEBUGPROCESS
- A process calls the Debug system procedure.
- A process starts another process by calling the NEWPROCESS, NEWPROCESSNOWAIT, PROCESS_CREATE_, or PROCESS_LAUNCH_ system procedure with the debug bit set.
- A process encounters a trap for which it does not have a trap handler.
- A process attempts to call an unresolved external routine.

After Inspect suspends program execution, it displays the status message and then prompts you for a command. Inspect might report incorrect information for programs that have been optimized. Therefore, if the current location in the program has been optimized in such a way that incorrect information might result, Inspect displays this warning message for debug events:

```
** Inspect warning 198 ** Results might be unexpected due to optimization
```

Breakpoints and Break Events

A breakpoint is a location in your program where you tell Inspect to suspend execution so you can examine the program's state and perhaps modify its variables. Inspect breakpoints can consist of three parts:

- Break location—where the break event should occur
- Break condition—under what conditions the break event should occur
- Break action—what to do after the break event occurs
- Break duration—how long the break exists

The condition and action parts are optional features that extend the capabilities of simple breakpoints.

A break event occurs when program execution reaches a breakpoint, provided that the conditions (if any) limiting that breakpoint are fulfilled. Once the break event occurs, Inspect puts the program into the hold state, reports the breakpoint, and then performs any break action associated with the breakpoint.

Setting Breakpoints

Inspect maintains a breakpoint list for each program you are debugging, and enables you to add breakpoints to or to remove breakpoints from the current program's breakpoint list. The time at which you set a breakpoint is called breakpoint definition.

Code Breakpoints

A breakpoint whose break location refers to object code (whether at the source or machine level) is called a code breakpoint. In Inspect, you specify the location of a code breakpoint using a code location. Code breakpoints are activated when the code specified by the code location is about to be executed.

When debugging on a TNS/R system, code breakpoints may be set at any location in non-accelerated or TNS/R native programs. However, when debugging accelerated programs, Inspect only allows TNS code breakpoints to be set at location that are memory-exact points. An attempt to set a TNS breakpoint at a location that is not a memory-exact will result in an error. Fore more information, see Section 16, Using Inspect With Accelerated Programs on TNS/R Systems.

When debugging PATHWAY requester programs, you can put code breakpoints in inactive scope units only at the entry point of the scope unit.

Note. Inspect does not limit the number of breakpoints you can set in a single program. However, after breakpoint ninety-nine, Inspect will stop numbering them. The total number of code breakpoints in all processes and PATHWAY servers running in a single processor is further limited; it is specified by BREAKPOINT_CONTROL_BLOCKS at system generation time. Consult your system operator or system manager for the maximums used in your system's processors.

The total number of code breakpoints in all PATHWAY requester programs under the control of a single TCP cannot exceed twenty.

Data Breakpoints

A breakpoint whose break location refers to a data item is called a data breakpoint. In Inspect, you specify the location of a data breakpoint using a data location. Data breakpoints are activated when the data word specified by the data location is stored to or (optionally) read from or changed. This detection of read access extends to the special, read-only data types provided by some languages (for example, P-relative arrays in TAL).

The default type for high-level data breakpoints is change (formerly write). The new default applies to both TNS and TNS/R systems. Data breakpoints are only reported if the value of the variable has changed; writes that store the same value already contained in the variable are not reported. You can have only one data breakpoint at a time in each program that you are debugging.

Note. Specific constraints apply to data breakpoints when executing accelerated programs on TNS/R systems. For more information, see Section 16, Using Inspect With Accelerated Programs on TNS/R Systems. In addition, data breakpoints are not valid in PATHWAY requester programs

---

Conditional and Unconditional Breakpoints

A breakpoint whose definition includes a break condition is called a conditional breakpoint. If a breakpoint's definition does not include a break condition, it is called an unconditional breakpoint. The conditionality of a breakpoint affects how Inspect processes the breakpoint, as discussed in the following subsection.

Processing Breakpoints

Whenever one of the programs you are debugging encounters a break location, Inspect begins processing the breakpoint at that break location. This is called breakpoint activation. If the breakpoint is unconditional, Inspect generates a break event immediately. On the other hand, if the breakpoint is conditional, Inspect does not generate a break event unless the specified break condition is met. Once the break event occurs, Inspect suspends the program containing the breakpoint.

After recording the current breakpoint, Inspect performs any break action associated with the breakpoint. If the break action concludes with a RESUME command, Inspect does not display the status messages and prompt as it normally would; instead, it resumes execution immediately.

Source Languages and Inspect

To support several source languages, Inspect uses generic concepts to describe a program and its contents. These concepts include:

- Scope of identifiers
- Scope units and scope paths
- Activation of scope units, including recreative activation
- Expressions
- Code locations
- Data locations

The availability and applicability of these concepts differ from language to language; however, Inspect presents them similarly whenever possible.

Scope of Identifiers

The concept of scope is central to several programming languages. The scope of an identifier determines the portion of the program in which that identifier is defined (also referred to as the domain of an identifier). All the languages that use the concept of scope provide some construct by which the programmer can define the boundaries of a specific scope domain. Each language has its own name for these constructs (for example, C calls them functions and COBOL85 calls them program units), but Inspect groups them all under the term scope unit.

Some languages even allow one scope unit to contain other scope units (COBOL85 and Pascal, for example). In this case, the domain of an interior scope unit is defined by the boundaries of its containing, or parent, scope unit.

Compilers for scope-dependent languages use the scope of an identifier to distinguish it from similarly named identifiers in other scope units. Inspect uses the scope of an identifier for exactly the same purpose.

Scope Paths

In Inspect, you define the scope of an identifier by specifying the scope path to that identifier. The scope path is the list of scope units containing the identifier, starting with the outermost and working down to innermost (the one containing the identifier). In some cases, the scope path consists of a single scope unit, while in other cases it might consist of ten or more scope units.

The general syntax you use to specify a scope path to Inspect is common to all languages:

```
#scope-unit [ .scope-unit ]...
```

As an example, examine this code fragment (written in a simple pseudolanguage):

```
scope-unit main begin
    ...
    scope-unit deep begin
        identifiers: a,b,c
        ...
        scope-unit deeper begin
            ...
            scope-unit deepest begin
                identifiers: x,y,z
                ...
            end; {end of deepest}
            ...
        end; {end of deeper}
        ...
    end; {end of deep}
    ...
end; {end of main}
```

According to the general syntax, the scope path for the identifiers A, B, and C is:

```
#main.deep
```

The scope path for the identifiers X, Y, and Z is:

```
#main.deep.deeper.deepest
```

Refer to the language-specific portions of this manual for language-specific syntax.

The Current Scope Path

If you do not specify a scope path when you refer to an identifier, Inspect assumes that the identifier is in the current scope path. You can set the current scope path using the SCOPE command. In addition, Inspect sets the current scope path whenever a debug event occurs. In this case, Inspect sets the current scope path to the scope path defining the scope unit in which the debug event occurred.

Using the code fragment from the previous subsection, assume that a breakpoint is set at the entry of the DEEPEST scope unit. When that breakpoint halts as a result of that breakpoint, Inspect sets the current scope path to:

```
#main.deep.deeper.deepest
```

To look at the value of the identifier X, you could then enter:

```
-SAMPLE-DISPLAY #main.deep.deeper.deepest.x
```

Alternatively, you can use the current scope path and simply enter:

```
-SAMPLE-DISPLAY x
```

Inspect will automatically qualify the identifier X using the current scope path.

Activation of Scope Units

Scope paths qualify and define specific scope units, but they are not related to the execution of a scope unit. Execution is controlled by the control flow in a program. When a program firsts begins, only the main scope unit is active. When that scope unit calls some other scope unit, the other scope unit becomes active as well. This second scope unit remains active until it returns control to the scope unit that called it.

When a scope unit is activated, space for its local variables is allocated. When it is deactivated (that is, when it returns control to its caller), its local variable space is deallocated. As a result, data breakpoints referring to local variables become useless when their scope unit is deactivated.

When debugging, you can examine the list of active scope units using the TRACE command. This command displays the call history from the scope unit currently being executed back to the main scope unit in the program. If a scope unit does not appear in the call history, it is called an inactive scope unit.

Note. Inspect permits a limited set of operations for inactive scope units. In an inactive scope unit, you can:

- Set or clear breakpoints.
- Display the attributes of an identifier.
- Display object code and source text (but not data values).

Recursive Activation

Recursion occurs when a scope unit calls itself or another active scope unit. Recursion causes two or more activations, or instances, of a single scope unit to appear in the call history. Because each such instance has its own local variables, a data reference to one of these variables is ambiguous.

To specify the particular activation to which a local data reference refers, you add an instance number to the end of the scope path. If you do not specify an instance number, Inspect assumes that the reference is to the most recent activation of the variable's parent scope unit. When you give an instance number (for example, 1,2,3,…), you can specify it relative to the most recent activation or the earliest activation, where 1 is the first, 2 is the second, and so on:

Zero (0)	Specifies the most recent activation.
Negative (-1,-2,-3,…)	Specifies the instance relative to the most recent activation: -1 is the second-most recent, -2 is the third-most recent, and so on.
Positive (1,2,3,…)	Specifies the instance relative to the first activation: 1 is the first, 2 is the second, and so on.

To examine an element as it exists during a particular instance, include an instance number when you specify the scope path. You can count from either direction using these conventions:

- Instance 1 is the least recent instance—that is, the oldest chronologically. Positive values count from the base of the stack toward the top.
- Instance 0 is the most recent instance—that is, the current scope path.
- Instance -1 is the next most recent—that is, the youngest chronologically. Negative values count from the top of the stack toward the base.

A scope instance is either a relative instance or an absolute instance. Both are useful; a user can easily specify the first occurrence of a scope of the last occurrence without knowing the number of instances. This example illustrates the difference between relative and absolute instances:

```
-PROGRAM-TRACE
Num Lang Location
  0  TAL #A.#42
  1  TAL #B.#57
  2  TAL #A.#101
  3  TAL #A.#4
  4  TAL #C.#67
  5  TAL #B.#49
  6  TAL #A.#78
  7  TAL #M.#3
```

The number in the leftmost columns is the scope ordinal. Scope 0 is always the most recent scope, or the current execution scope. Its caller is scope ordinal 1; the caller of scope 1 is scope ordinal 2 and so on. Scope ordinals number all active scopes; scope instances number all active instances of the same scope.

In the previous example, A at statement 42 is relative instance 0, A at 101 is relative instance -1, A at statement 4 is relative instance -2 and A at statement 78 is relative instance -3. Alternatively, A at statement 78 can be referred to as absolute instance 1, A at statement 4 as absolute instance 2, A at statement 101 as absolute instance 3 and A at statement 42 as absolute instance 4.

Scope Ordinal	**Scope Name**	**Line Number**	**Relative Instance**	**Absolute Instance**
0	A	42	0	4
1	B	57	0	2
2	A	101	-1	3
3	A	4	-2	2
4	C	67	0	1
5	B	49	-1	1
6	A	78	-3	1
7	M	3	0	1

Note. You can specify instances for active scope units only. In addition, an instance must already exist—that is, you cannot refer to the tenth instance of a local variable when only eight activations of its parent scope unit are in the call history.

Expressions

An Inspect expression is a list of operands and operators which, when evaluated, result in a number or string. The operators you can use are based on the source language of the program you're debugging.

Note. Inspect does not permit function calls as operands

Code Locations

A code location is a symbolic reference to a specific location in the object code of a program. Although the syntax used to specify a code location varies slightly for each source language, the general syntax is the same for all languages.

```
{ scope-path                          }
{                                     } [ code-offset [1] [2] [3]
{ [ scope-path. ] code-reference }


code-reference:  one of

   scope-unit [ FROM source-file ]
   label [ FROM source-file ]
   #line-number [ (source-file) ]


code-offset:

  { + | - } num-units [ code-unit ]
  { + | - } num-units [ code-unit ]
  { + | - } num-units [ code-unit ]


code-unit:  one of


   INSTRUCTION[S]   STATEMENT[S]   VERB[S]
```

scope-path

specifies the scope path to the scope unit containing the code location. When used alone (the first option in the main syntax), *scope-path* specifies the primary entry point of the last scope unit named in the scope path.

[*scope-path*.] *code-reference*

specifies a named or numbered location in the scope unit defined by the given scope path (or the current scope path if no scope path is given). The exact form of *code-reference* varies from language to language, but options shown in the previous diagram are generally available.

scope-unit [FROM *source-file*]

specifies the primary entry point of the scope unit. *scope-unit*must be the same as the last scope unit named in *scope-path* (or in the current scope path).

The FROM clause identifies the scope unit by the source file in which it is found; therefore, Inspect can distinguish between scope units that have the same name, but reside in different modules.

`label [ FROM source-file ]`

specifies the statement following a given label in the source code.

The FROM clause identifies the label by the source file in which it is found; therefore, Inspect can distinguish between labels that have the same name.

`#line-number [ (source-file) ]`

specifies the statement starting at a given line number in the source file. If no statement begins at the specified line number, Inspect issues this warning:

```
A subsequent line number is assumed: line-num
```

Inspect then uses the statement starting at line *line-num*. If more than one statement begins on the specified line, Inspect uses the start of the first statement.

The (*source-file*) option identifies *#line-number* by the source file in which it is found. You need to use this option only if the source code for the given scope unit is in more than one file.

code-offset

specifies an offset from the code location defined by the preceding options. A positive offset (+) denotes code following the specified code location; a negative offset (-) denotes code preceding the specified code location. The amount to offset is specified by a given number of units. If you omit the unit specifier, Inspect selects a default unit depending upon the current source language. This default unit will always be a statement.

Usage Considerations

- Using source-file with *#line-number*

  Inspect assumes that the source code for a scope unit is in a single file. When it is not in a single file, using *#line-number* alone to identify a code location will not work if the line number you want isn't in the first file. To direct Inspect to the correct file, use the *source-file* option shown in the general syntax.

- Optimized Statements

  If optimization has removed the code for a statement and Inspect encounters the statement, Inspect displays the message:

```
**** ERROR 197 **** Location deleted due to optimizations
```

  You cannot set a breakpoint for a deleted statement.

Data Locations

A data location is a symbolic reference to a data item within a program. Although the syntax used to specify a data location varies for each source language, the general syntax is the same for all languages.

```
[ scope-path [ (instance) ] . ] data-reference
```

scope-path

specifies the scope path to the scope unit containing the data item.

instance

specifies a specific activation of the scope unit containing the data item.

data-reference

specifies a data item using the syntax of the source language.

Usage Considerations

These considerations are valid for all source languages.

- Modifying and Displaying Program Data

  When program execution begins, the compilers add code in front of the first program statement which initializes the run-time library and sets up the variables. When you inspect a program, step to the first statement of your code before accessing variables.

- How Inspect Displays Names of Data Items

  When Inspect displays the name of a data item (for example, when you direct Inspect to display the value of a data item), Inspect uses a form similar to Pascal or TAL. Assume you request Inspect to display the value of this COBOL data item:

  ```
  hours OF overtime OF weeks-work (3)
  ```

  Inspect displays its name as follows:

  ```
  WEEKS-WORK[3].OVERTIME.HOURS
  ```

- Specifying Subscript Ranges

  When referring to arrays, Inspect enables you to specify a subscript range instead of a single subscript value. For example, suppose the following data item occurs in a COBOL program:

  ```
  01 lumber-table.
     03 thickness OCCURS 2 TIMES.
        05 width OCCURS 6 TIMES.
           07 price PICTURE 99V99.
  ```

You can display the value of PRICE for the fourth through sixth occurrences of WIDTH of the second occurrence of THICKNESS by entering the command:

```
-COBOBJ-DISPLAY price(2,4:6)
LUMBER-TABLE.THICKNESS[2].WIDTH[4].PRICE= 1.89
LUMBER-TABLE.THICKNESS[2].WIDTH[5].PRICE= 2.09
LUMBER-TABLE.THICKNESS[2].WIDTH[6].PRICE= 2.29
```

You can modify the value of PRICE for the third through fifth occurrences of WIDTH of the first occurrence of THICKNESS by entering the command:

```
-COBOBJ-MODIFY price(1,3:5) = 1.29,1.49,1.69
```

3 Inspect Command Overview

For detailed descriptions of the high-level and low-level Inspect commands, see Section 6, High-Level Inspect Commands, and Section 7, Low-Level Inspect.

Entering Inspect Commands

Inspect processes commands on a line-by-line basis. When it reads a line, whether from the Inspect command terminal, the INSPLOCL file, the INSPCSTM file, or an OBEY file, Inspect expects that line to be either:

- A blank line
- A command list

A command list consists of either a single Inspect command or a list of Inspect commands separated by semicolons. If a command list includes an alias name, Inspect substitutes the alias's replacement text for its name.

If a command list is too long to fit on a single line, you can continue it on the next line by ending the first line with an ampersand (&). For example, assume you enter this:

```
-PRG-DISPLAY height, width, &
-PRG-depth, (height*width*depth)
```

The effect is that same as this:

```
-PRG-DISPLAY height, width, depth, (height*width*depth)
```

Using the ampersand to continue the command list, you can create command lists of up to 512 characters, including the space required to expand any aliases in the command list.

There are two cases for pressing the BREAK key:

- When entering an Inspect command

  Pressing BREAK returns you to the TACL prompt. Enter PAUSE to return to Inspect.

- When fixing a command (within the FC command)

Pressing BREAK returns you to the Inspect prompt.

If you press the BREAK key when entering a command (even if you are on a continuation line), Inspect retains the accumulated command list, except for commands on the line that you were entering. However, Inspect does not interpret the command list; instead, it prompts for another command. Consequently, if you find that you have made a mistake in a line, you can use the FC command to correct all but the last input line of the command list and then reissue it.

△ **Caution.** A COMMENT (“- -”) command causes Inspect to ignore any commands that follow it in the command list.

Debugging the Current Program

The basic debugging commands that Inspect offers affect only one program: the current program. Table 3-1 lists the basic debugging commands that fall into three categories:

- Commands that control program execution
- Commands that access program information
- Commands that simplify program debugging

Table 3-1. Commands for Debugging the Current Program (page 1 of 2)

Command	Description
Commands that Control Program Execution	
BREAK	Sets one or more breakpoints in the current program or display current breakpoints.
CLEAR	Clears one or more breakpoints in the current program.
HOLD	Suspends the execution of a program on the program list, placing it in the hold state.
RESUME	Reactivates a suspended program, changing its state from hold to run.
STEP	Resumes execution of the current program at the point where it was last suspended, and then suspends execution after the program has executed a certain number of units.
STOP	Stops one or more programs and removes them from the program list.
Commands that Access Program Information	
DISPLAY	Formats and displays the value of variety of items, including constants, expressions, code, and data.
INFO	Displays internal information about various entities in the current program.
MATCH	Searches for scope-unit names or other identifiers in the current program.

Table 3-1. Commands for Debugging the Current Program (page 2 of 2)

Command	Description
MODIFY	Changes the value of a data item or register in the current program.
SAVE	Creates a save file of the current program, including its extended segments.
SOURCE	Displays source code.
TRACE	Displays the call history for the current program location.
Commands that Simplify Program Debugging	
LIST BREAKPOINT	Displays one or all breakpoints defined in the current program.
SCOPE	Changes or displays the current scope path.

Managing Multiple Programs

Inspect enables you to debug more than one program in a single Inspect session. Table 3-2 lists the commands that you use to manage multiple programs.

Table 3-2. Commands for Managing Multiple Tables

Command	Description
ADD PROGRAM	Adds a process or save file to the program list for the current Inspect session.
PROGRAM	Adds a process or save file to the program list for the current Inspect session.
LIST PROGRAM	Displays the list of programs being debugged.
SELECT PROGRAM	Selects a program from the program list as the current program.

Managing Source Files

When you debug a program written in a high-level programming language, you often need to examine the program's source code to pinpoint a bug. However, if the source code is moved between the time that the program was compiled and the time that you debug it, Inspect cannot retrieve the source code from its new location automatically. Consequently, Inspect provides several commands that enable you to inform Inspect of changes to the location of source code, as listed in Table 3-3 on page 3-4.

Table 3-3. Commands for Managing Source Files

Command	Description
ADD SOURCE ASSIGN	Adds a source assignment to the current program's source assignment list.
DELETE SOURCE ASSIGN	Removes one or all source assignments from the current program's source assignment list.
DELETE SOURCE OPEN	Closes one or all source files that Inspect has opened as the result of previous SOURCE commands.
LIST SOURCE ASSIGN	Displays the source assignments from the source-assignment list for the current Inspect session.
LIST SOURCE OPEN	Displays the names of all source files that Inspect has opened as the result of previous SOURCE commands.
SELECT SOURCE SYSTEM	Directs Inspect to retrieve source files from a specific system when a source file name in the current program does not explicitly include a system name.
SOURCE ASSIGN	Sets or displays source assignments from the source-assignment list for the current Inspect session.

Entering and Editing Inspect Commands

Like TACL, Inspect maintains a history buffer that records the commands you enter. Inspect provides commands that enable you to display, edit, and reissue commands in this buffer. Inspect also provides commands that enable you to edit and reissue commands that define aliases, function keys, and breakpoints. Table 3-4 lists these commands, in addition to others that affect or explain how you enter Inspect commands.

Table 3-4. Commands for Entering and Editing Inspect Commands (page 1 of 2)

Command	Description
FA	Enables you to retrieve, edit, and reissue an existing alias definition.
FB	Enables you to retrieve, edit, and reissue an existing breakpoint definition.
FC	Enables you to retrieve, edit, and execute a command line in the history buffer.
FK	Enables you to retrieve, edit, and reissue an existing function-key definition.
HELP	Displays information regarding the syntax and usage of Inspect commands and command options.
HISTORY	Displays the most recently executed command lines.

Table 3-4. Commands for Entering and Editing Inspect Commands (page 2 of 2)

Command	Description
LIST HISTORY	Displays a portion of or the entire history buffer.
SELECT LANGUAGE	Changes the current source language, therefore changing the acceptable syntax of language-dependent entities.
XC	Reexecutes a command line in the history buffer.

Customizing an Inspect Session

Inspect provides several commands that enable you to customize your Inspect session. Table 3-5 lists these commands.

Table 3-5. Commands for Customizing an Inspect Session

Command	Description
ADD ALIAS	Adds an alias to the alias list for the current Inspect session.
ADD KEY	Adds a function-key definition to the function-key list for the current Inspect session.
ALIAS	Adds an alias to the alias list or displays aliases for the current Inspect session.
DELETE ALIAS	Removes one or all aliases from the alias list for the current Inspect session.
DELETE KEY	Removes one or all function-key definitions from the function-key list for the current Inspect session.
ENV	Displays the current settings of the Inspect environment parameters, including the parameters controlled by the SELECT command.
LIST ALIAS	Displays one or all aliases from the alias list for the current Inspect session.
LIST KEY	Displays one or all function-key definitions from the function-key list for the current Inspect session.
SET	Changes the status of one of the settable Inspect parameters.
SHOW	Shows the status of one or all the Inspect parameters controlled by the SET command.
SYSTEM	Sets the default system for expansion of any file names.
VOLUME	Sets the default volume and subvolume for expansion of any file names.

Managing an Inspect Session

Inspect provides several commands that enable you to manage and control your Inspect session. Table 3-6 lists these commands.

Table 3-6. Commands for Managing and Inspect Session

Command	Description
COMMENT (OR “- -”)	Directs Inspect to ignore the remainder of the command line, therefore enabling you to add remarks in a LOG file, an OBEY file, or the INSPLOCL or INSPCSTM configuration files.
EXIT	Stops the Inspect process, therefore terminating the Inspect session.
HIGH	Causes Inspect to change from low-level to high-level command mode or to remain in high-level if you are in high level command mode.
IF	Provides conditional execution of an Inspect command or alias.
LOG	Records the session input, output, or both input and output on a permanent file.
LOW	Causes Inspect to change from high-level to low-level command mode or to remain in low-level if you are in low-level command mode.
OBEY	Causes Inspect to read commands from a specified file.
OUT	Directs the output listing to a specified file.
PAUSE	Suppresses Inspect prompts until a debug event occurs in any of the programs on the program list.
TERM	Changes which terminal or process is the Inspect command terminal.

Simplifying an Inspect Session

Inspect provides several commands that simplify working with Inspect. Table 3-7 lists these commands.

Table 3-7. Commands for Simplifying an Inspect Session (page 1 of 2)

Command	Description
ALIAS	Adds a name or a command string, to the alias list for the current Inspect session.
BREAK	Clears one or more breakpoints in the current program.
FILES	Shows the status of files that have been opened by the current program. The FILES command is a synonym for the INFO OPENS command.
HISTORY	Displays the most recently executed command lines.I

Table 3-7. Commands for Simplifying an Inspect Session (page 2 of 2)

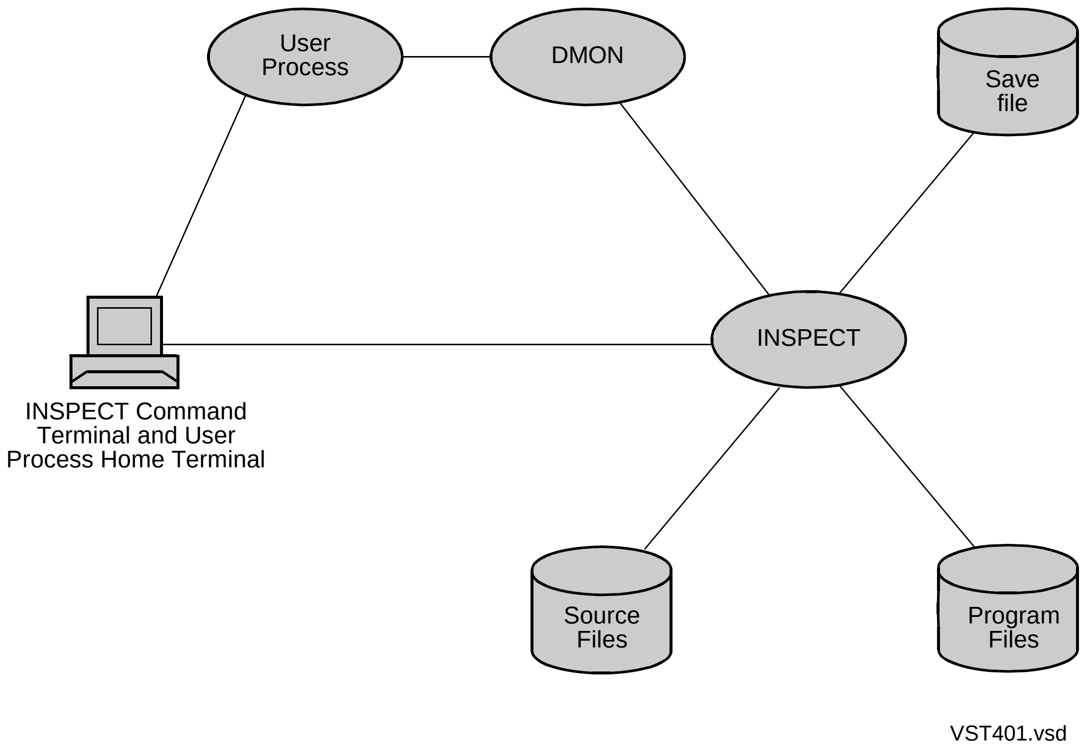
Command	**Description**
IDENTIFIER	Displays information about the internal characteristics of a given data location or of all data locations in one or more scope units. The IDENTIFIER command is a synonym for the INFO IDENTIFIER command.
KEY	Adds a function-key definition or displays one or all function-key definitions in the function-key list for the current Inspect session.
OBJECT	Displays information about the current program's object file.
OPENS	Shows the status of files that have been opened by the current program. It is a synonym for the INFO OPENS command.
PROGRAM	Displays the programs in the program list or selects a program as the current program.
SOURCE ASSIGN	Sets or displays source assignments from the source-assignment list for the current Inspect session.
SOURCE OFF	Disables automatic source display at each event.
SOURCE ON	Enables automatic source display at each event.
SOURCE OPEN	Displays the names of the files that are currently open as a result of previous SOURCE commands. The SOURCE OPEN command is a synonym for the LIST SOURCE OPEN command.
SOURCE SYSTEM	Directs Inspect to retrieve source files from a specific system when the current name of a source file in the current program does not explicitly include a system name. SOURCE SYSTEM is a synonym for the SELECT SOURCE SYSTEM command.

4 Debugging Processes and Save Files

Inspect in the Guardian Environment

Figure 4-1 shows the various components involved when you debug processes and save files in the Guardian environment.

Figure 4-1. Inspect in the Guardian Environment



The Inspect process retrieves symbol information from the program files of the processes and save files, and retrieves source code from the source files. DMON provides the Inspect execution control services for process debugging.

These control services include:

- Setting and clearing breakpoints (non save files)
- Providing execution status information
- Retrieving machine code and data values for display
- Modifying data values

Note. To debug processes, $IMON must be running on the system hosting the Inspect command terminal, that is the system that you want to debug the process on and DMON must be running on the same CPU. If $IMON is not on the system hosting the command terminal and $DMnn is not on the same processor, you will invoke Debug, to examine save files, however, neither $IMON nor $DMnn is required. $IMON is usually started at the same time as the operating system. If no $IMON is running on your system, contact your system manager.

Command and Home Terminals

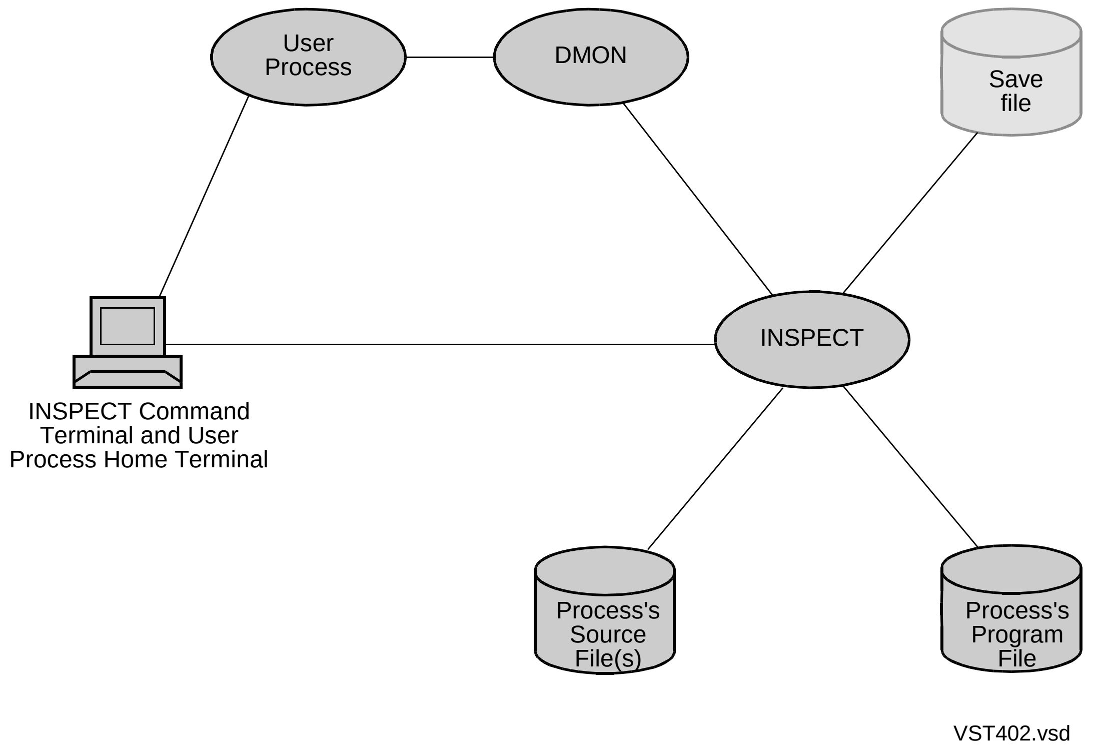
Command terminals are terminals where system managers, developers, and programmers interact with system-level utility programs. When debugging processes or examining save files, only one command terminal is involved: the Inspect command terminal.

Home terminals are terminals where a process is started. Each process has a home terminal; if a process is started by another process (instead of from a terminal), it inherits the home terminal of its creator. Inspect uses the home terminals of the process to determine which terminal it should use as its command terminal.

Debugging Processes

Figure 4-2 highlights the components involved in process debugging.

Figure 4-2. Debugging Processes



Before you can begin to debug a process with Inspect, you must first ensure that the operating system will select Inspect, not Debug, as the debugging tool for the process. The software makes this selection based upon the debugging attributes of the process. The following subsections introduce these attributes and describe how you can control their values.

The Debugging Attributes of a Process

Each process has two debugging attributes, INSPECT and SAVEABEND. The software sets the values of these attributes for a process when it starts the process. The Inspect attribute determines which debugging tool (Inspect or Debug) to use for the process.

Specifically, on a TNS/R system, these debugger selection rules apply:

INSPECT Attribute	**Meaning**
INSPECT ON	Selects either Inspect or Visual Inspect as the debugging tool for the process. To use Visual Inspect, you must have a client connection set up before the debugger is invoked. If neither Visual Inspect nor Inspect is available, Debug is selected as the debugger of last resort. Only Visual Inspect (not Inspect) can be used to debug TNS/R native position-independent code (PIC) program files.
INSPECT OFF	Selects Debug as the debugging tool for the process.

The SAVEABEND attribute controls whether a save file is created automatically if the process abends, and the SAVEABEND attribute interacts with the INSPECT attribute.

Specifically, on a TNS/R system, these rules apply for the SAVEABEND attribute:

SAVEABEND Attribute	**Meaning**
SAVEABEND ON	Specifies that a save file should be created if the process abends. Setting SAVEABEND ON also sets INSPECT ON.
SAVEABEND OFF	Specifies that a save file should not be created if the process abends.

For more information about debugger selection and debugging attributes on a TNS/E system see Section 18, Using Inspect on a TNS/E System.

The system software sets the debugging attributes of a process when it starts the process. To select the proper values, the operating system compares these three sets of attribute values:

- The attribute values found in the program file.
- The attribute values of the creator of the process.
- The attribute values specified in the call that creates the process. TACL specific attribute values exist when the process is created based on the TACL SET INSPECT command and the run line.

If the Inspect attribute of any of these is ON, then the system software sets the process's INSPECT attribute to ON. If the SAVEABEND attribute of any of these is ON, then the system sets the process's SAVEABEND attribute to ON.

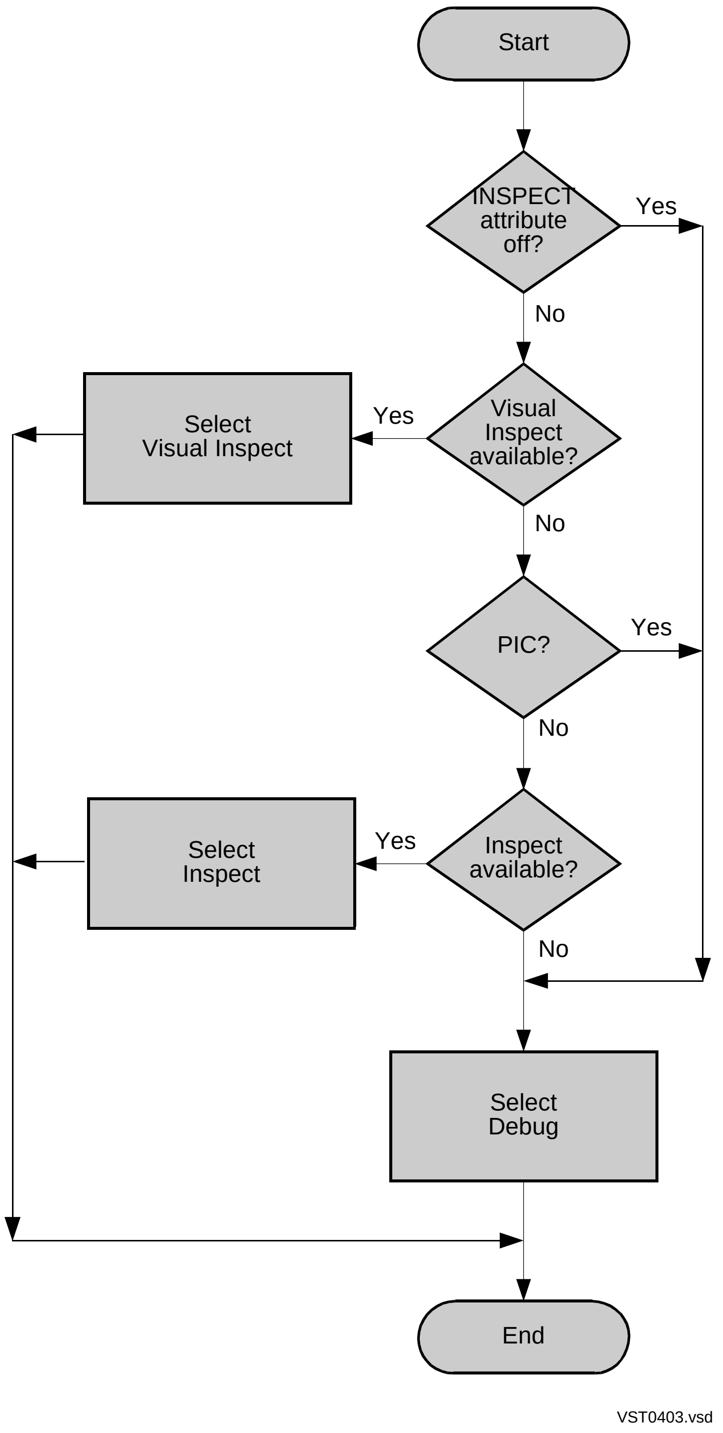
Note. If the creator of the process does not have read access to the program file, the system software sets both the Inspect and the SAVEABEND attributes to OFF.

Figure 4-3 on page 4-6, debugger selection criteria are defined as:

Criteria	**Meaning**
INSPECT attribute on?	The setting for INSPECT is set ON for the process you will debug (set with TACL, the compiler, or the linker).
Visual Inspect session?	You have started Visual Inspect and have connected to the NonStop host on which the process to be debugged will run. The user ID of the process must match the user ID that was used to log on to Visual Inspect.
Inspect available?	The Inspect subsystem (IMON, DMON, $DM*nn*) is running, and the Inspect command-line interface is available.

Figure 4-3 on page 4-6 shows the debugger selection process on a TNS/R system. The selection rules are the same for TNS programs and for those TNS/R native programs that are not PIC (Position-Independent Code). Note, however, that Inspect cannot be used to debug TNS/R PIC programs.

Figure 4-3. Debugger Selection on a TNS/R System



On a TNS/R system, the precedence of debuggers is as follows:

Process Type	INSPECT Attribute	Debugger Precedence
TNS	INSPECT ON	Visual Inspect, Inspect, Debug
TNS	INSPECT OFF	Debug
TNS/R Native non-PIC	INSPECT ON	Visual Inspect, Inspect, Debug
TNS/R Native non-PIC	INSPECT OFF	Debug
TNS/R Native PIC	INSPECT ON	Visual Inspect, Debug
TNS/R Native PIC	INSPECT OFF	Debug

Note that Visual Inspect can only be selected when a matching client connection already exists. If the Inspect subsystem is not available, the debugger of last resort is the TNS/R system debugger, Debug.

Debugging Attributes in a Program File

Either Binder (for TNS program files) or the native linker (for TNS/R program native files) sets the INSPECT and SAVEABEND attribute values of a program file when it creates that program file. You can control the values set by Binder or the native linker by using compiler directives, the BIND process's SET INSPECT and SET SAVEABEND commands, or native linker options, as appropriate.

On a TNS/R system, the nld native linker sets INSPECT ON and SAVEABEND OFF by default.

In most source languages, the compiler directives are called INSPECT and SAVEABEND, but consult the appropriate reference manual for their exact form. For more information about the SET INSPECT and SET SAVEABEND commands, see the *Binder Manual* (for TNS or TNS/R systems), the *nld Manual* and *noft Manual* (for PIC code on TNS/R systems), or the *ld and rld Reference Manual* (for TNS/E systems).

Debugging Attributes of a Process's Creator

Because the creator of a process is also a process (remember, the command interpreter is a process too), the system selects the INSPECT and SAVEABEND attribute values for the creator.

Debugging Attributes in the Call to Start a Process

There are several ways to start a process:

- The command interpreter RUN[D] command
- The TACL #NEWPROCESS built-in function
- The system procedure calls NEWPROCESS, NEWPROCESSNOWAIT, PROCESS_CREATE_, and PROCESS_LAUNCH_

RUN[D] and #NEWPROCESS. When you start a process using the RUN[D] command or the #NEWPROCESS built-in function, the Inspect state of the command interpreter session determines the attribute values of the start-up command. The Inspect state has one of three values: OFF, ON, or SAVEABEND. These state values correspond to these debugging attribute values:

Inspect State	**Debugging Attribute Values**
OFF	INSPECT OFF, SAVEABEND OFF
ON	INSPECT ON, SAVEABEND ON
SAVEABEND	INSPECT ON, SAVEABEND ON

You can control the value of the Inspect state using the SET INSPECT command, or you can override the default state value in the RUN command itself with the Inspect run-option. For more information, see the *TACL Reference Manual.*

NEWPROCESS, NEWPROCESSNOWAIT, PROCESS_CREATE_, and PROCESS_LAUNCH_. These system procedure calls enable an existing process to create a process. All these procedures include a parameter that contains bits corresponding to the INSPECT and SAVEABEND debugging attributes for the call. To find out how to set these bits, see the *Guardian Procedure Calls Reference Manual.*

Preparing and Configuring for Process Debugging

To use all the symbolic debugging capabilities of Inspect, you must ensure that your program file includes symbol information. Consequently, you must include the SYMBOLS compiler directive in the source code or on the command line when you compile the source code.

Starting a Debugging Session

A process debugging session can start in several ways, as shown in the following list. The first two methods shown start a process in the hold state. All the other methods put a running process in the hold state.

- You enter a command interpreter RUND command, starting the process in the hold state.
- An existing process invokes the NEWPROCESS, NEWPROCESSNOWAIT, PROCESS_CREATE_, or PROCESS_LAUNCH_ procedure with the debug bit set, starting the process in the hold state.
- You enter a command interpreter DEBUG command, putting a running process in the hold state.
- You use the TACL #DEBUGPROCESS built-in function, putting a running process in the hold state.

- A process invokes the DEBUGPROCESS procedure, putting another process in the hold state.

Note. A process must exit out of system code before it enters the HOLD state.

- The process itself invokes the operating system software's Debug procedure, putting itself in the hold state.
- The process encounters a trap for which it does not have a trap handler, putting the process in the hold state.

RUND, NEWPROCESS, NEWPROCESSNOWAIT, PROCESS_CREATE_, or PROCESS_LAUNCH_

When you start a process using the TACL RUND command, or when an existing process starts a process using the NEWPROCESS, NEWPROCESSNOWAIT, PROCESS_CREATE_, or PROCESS_LAUNCH_ procedure, the operating system starts the process in the hold state. The system then starts the debugging tool specified by the Inspect attribute of the process on the process's home terminal. For more information about debugging attributes and how the operating system sets them, see The Debugging Attributes of a Process on page 4-4.

When entering the RUND command, you can ensure that the operating system software selects Inspect as the debugging tool with the Inspect run-option:

```
17> RUND program-file /INSPECT ON/
```

You don't need to include this option if the Inspect attribute of the program file is ON, or if the Inspect state of the command interpreter session is ON or SAVEABEND (you can check the Inspect state using the command interpreter SHOW INSPECT command).

When invoking NEWPROCESS, NEWPROCESSNOWAIT, PROCESS_CREATE_, or PROCESS_LAUNCH_, an existing process can ensure that the operating system selects Inspect as the debugging tool by setting the appropriate bit of the *flags* parameter in the call. The process does not need to set this bit if the Inspect attribute of the program file is ON.

DEBUG Command, #DEBUGPROCESS, and DEBUGPROCESS

When you enter the command interpreter DEBUG command or use the TACL #DEBUGPROCESS built-in function, or when a process invokes the DEBUGPROCESS procedure, a debug event occurs for the specified process.

The operating system then starts the debugging tool specified by the Inspect attribute of the process on the process's home terminal.

Note. The process enters the hold state as soon as it completes the machine-code instruction that was executing when the debug event occurred. If the process has invoked a procedure, it will not enter the hold state until the procedure returns control to the process.

These three methods of starting an Inspect session can direct Inspect to use a terminal other than the process's home terminal as the Inspect command terminal. Selecting a different command terminal is useful when the process you're going to debug requires exclusive or extensive use of its home terminal, or if it owns the BREAK key.

Note. Changing to a terminal other than your home terminal changes the home terminal of the process being debugged.

Debug Procedure and Unhandled Traps

When a process invokes the Debug procedure or encounters a trap for which it has not established a handler, a debug event occurs for that process. The operating system then starts the debugging tool specified by the Inspect attribute of the process on the process's home terminal.

Guidelines for Debugging a Process

The following subsections offer guidelines that make the task of debugging processes easier.

Debugging with Two Terminals

If the process you are debugging requires exclusive or extensive use of its home terminal, or if it owns the BREAK key (see “Break Feature” in the *Guardian Programmer’s Guide*), you can direct Inspect to use a terminal other than the process's home terminal in a variety of ways:

- You can enter an Inspect TERM command.
- You can use the TERM clause of the command interpreter DEBUG command.
- You can use the TERM clause of the TACL #DEBUGPROCESS built-in function.
- A process specifies the term parameter when invoking the DEBUGPROCESS procedure.

The first method changes the Inspect command terminal after the Inspect session has begun, while the others redirect the Inspect command terminal just as the session begins.

Pressing the BREAK Key

Pressing the BREAK key on the Inspect command terminal has various results, depending on the status of the session at the moment you press BREAK:

- If the current program is in the run state, pressing the BREAK key causes Inspect to issue a prompt. Inspect does not, however, place the current program in the hold state: you must enter a HOLD command.
- If you have paused Inspect (using the high-level PAUSE command), pressing the BREAK key causes Inspect to issue a prompt.
- If Inspect has prompted you and is awaiting a command, pressing the BREAK key causes the command interpreter to issue a prompt. In this case, Inspect is not paused. Consequently, Inspect and the command interpreter will compete for control of the terminal until you enter a command interpreter PAUSE command.
- If Inspect is displaying information, pressing the BREAK key causes Inspect to stop the display and issue a prompt.
- If a process running on the Inspect command terminal has ownership of the BREAK key, pressing BREAK has whatever effect the process assigns to it. If this process stops while Inspect is suspended by a PAUSE command, the BREAK key cannot be used to activate Inspect.

  You can activate the Inspect process by forcing a debug event. Enter the command interpreter command DEBUG, giving the process that owns BREAK as the process to debug, or enter the command interpreter command RUND to start debugging another process. Once Inspect regains controls of the terminal, use the TERM command to change the Inspect command terminal to avoid further BREAK problems.

Freeing the Inspect Command Terminal

If you need to communicate with some process other than Inspect or the process you are debugging, you can free the Inspect command terminal temporarily using the low-level PAUSE command. Follow these steps:

1. Enter the LOW command to go into low-level Inspect (if you are not there already).
2. Enter the PAUSE command:

```
_PRG_P pause-time
```

   The *pause-time* parameter specifies the number of centiseconds (hundredths of a second) you want Inspect to pause.

3. Press the BREAK key, therefore returning control to the command interpreter.

When you have finished using the command interpreter, enter the command interpreter PAUSE command and Inspect will again control the terminal. If Inspect has already used up its pause time, it will start prompting you again, perhaps before you pause the command interpreter process.

Ending a Debugging Session

If you want your process to continue running after you end the debugging session, use the EXIT command or the RESUME * EXIT command. If you want to terminate your process, use the STOP command.

When you enter the EXIT command, Inspect terminates the Inspect session, but it leaves the programs that you were debugging in their current states. Consequently, you should use the EXIT command only after you have cleared all breakpoints in all programs and resumed execution of any programs in the hold state. The RESUME * EXIT command performs these cleanup tasks for you.

When you enter the RESUME * EXIT command, Inspect clears all breakpoints from all programs on the program list and resumes execution of any programs in the hold state. It then terminates the Inspect session. If you attempt to leave an active process, all EXIT commands prompt you for a confirmation.

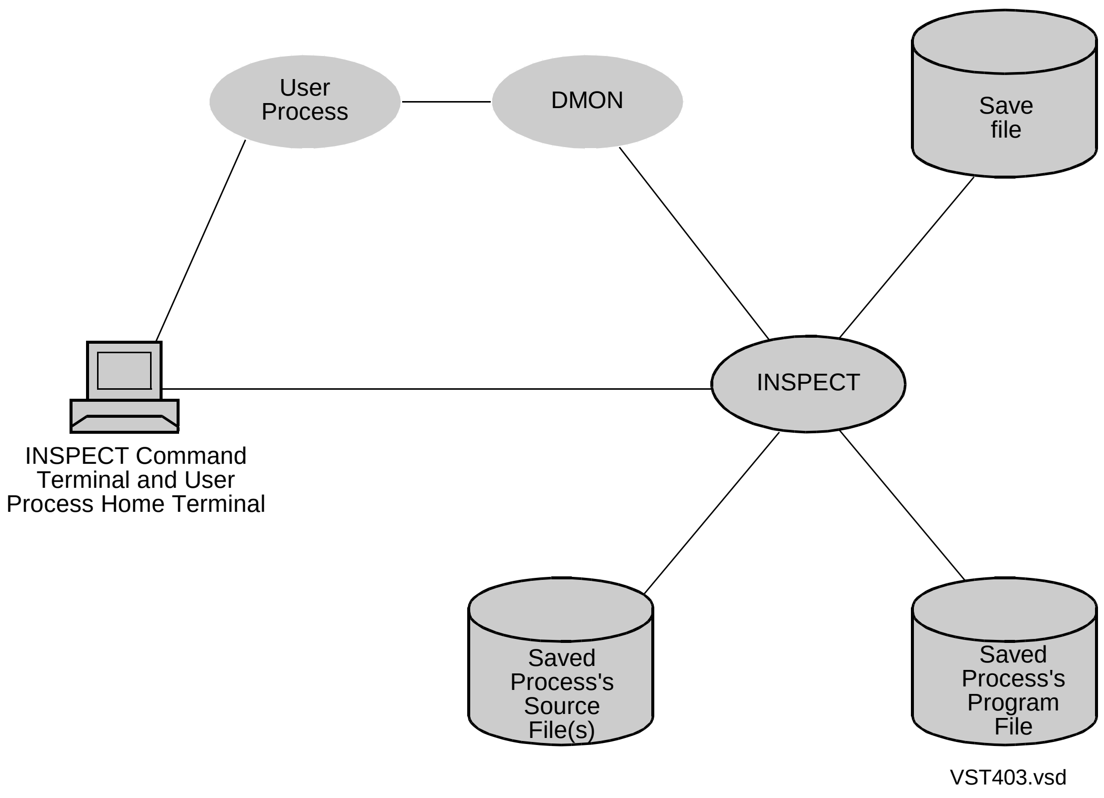
When you enter the STOP command, Inspect stops the process. If it was the only program in the program list, Inspect ends the session as well.

If you have paused the command interpreter on the Inspect command terminal, you might have to press the BREAK key to signal the command interpreter that it should prompt again.

Examining Save Files

Figure 4-4 highlights the components involved when examining save files.

Figure 4-4. Examining Save Files


User Process
DMON
Save file
INSPECT
INSPECT Command Terminal and User Process Home Terminal
Saved Process's Source File(s)
Saved Process's Program File
VST403.vsd

The Inspect process provides all the necessary services when you are examining a save file; neither IMON nor DMON is involved.

To start examining a save file, enter the command interpreter Inspect command:

```
5> INSPECT
```

When the Inspect session starts, Inspect displays its banner line and then issues an Inspect prompt. To retrieve a save file for examination, enter an ADD PROGRAM command specifying the name of the save file. Inspect reads the file from disk and delivers a status line showing the state of the process when the save file was created.

You can then examine the values of variables, display the call history, or view source files. The Inspect commands that you cannot use when examining a save file are:

BREAK
CLEAR
HOLD
MODIFY

RESUME
SAVE
SELECT DEBUGGER DEBUG
STEP

If you attempt to use one of these commands, Inspect displays the message:

```
** Inspect error 21 ** Invalid operation on a saved program
```

The TYPE clause of the DISPLAY command may yield different results for a save file than for the equivalent running process. These types rely on dynamic information that may have changed since the save file was created:

- PROCESS HANDLE
- SSID
- TRANSID

These types may be affected if the save file is moved to another node on the network:

- USERID
- USERNAME

These types are affected if the save file is removed from one network to another:

- CRTPID
- DEVICE
- FILENAME
- FILENAME32
- SYSTEM

Type TIMESTAMP will use the daylight saving time table and Greenwich Mean Time offset of the current node to format and display the local civil time.

You can transport save files between systems for examining problems. It may be necessary to point Inspect at the object file(s) for the process. This can be done using the CODE and LIB clauses of the ADD PROGRAM command. It may also be necessary to point to the locations of source files, which is described in the SOURCE ASSIGN command portion of Section 6, High-Level Inspect Commands.

When transporting save files to systems running different releases of Inspect, it may be necessary to transport the Inspect component of the Inspect sub-system for use in examining the save file if the installed version of Inspect does not recognize the version of the save file. When this is done, the Inspect can only be used for the purpose of examining save files.

When you are finished examining a save file, you use the STOP command to notify Inspect that you are done with the save file.

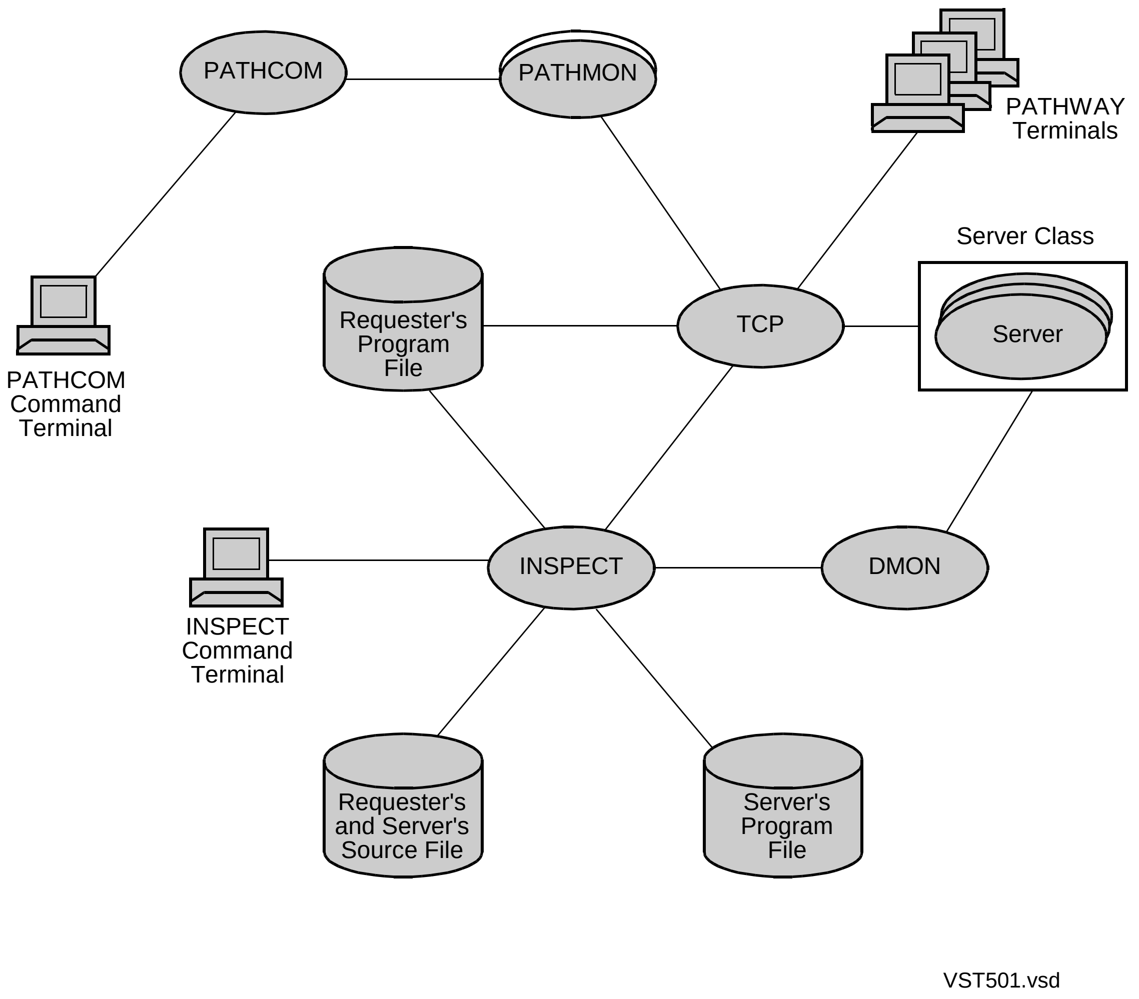
If you have been using Inspect solely to examine save files, a series of STOP commands (or STOP *) closes all the save files, but Inspect continues to execute. You can use the EXIT command at any time and Inspect closes all save files and exits. You will only receive a confirmation that you want to exit if you have active processes.

5 Debugging PATHWAY Applications

Inspect in the PATHWAY Environment

Figure 5-1 shows how Inspect interacts with the PATHWAY system to provide debugging facilities for PATHWAY applications.

Figure 5-1. Inspect in the PATHWAY Environment



The Inspect process retrieves symbol information from the program files of the requester programs and servers, and retrieves source code from the source files of the requester programs and servers. DMON provides the Inspect execution control services for server debugging, while the TCP, in addition to its normal functions in the PATHWAY environment, provides these services for requester program debugging.

Note. To debug PATHWAY applications, $IMON must be running on the system hosting the application. $IMON is usually started at the same time as the operating system. If no $IMON is running on your system, contact your system manager.

Application, Command, and Home Terminals

Application terminals are terminals where users interact with PATHWAY applications. TCPs manage these terminals and PATHWAY requesters (that is, a TCP executing a requester program) control them. Sometimes they are called PATHWAY terminals.

Command terminals are terminals where system managers, developers, and programmers interact with system-level utility programs. There are two command terminals involved in PATHWAY application debugging: the PATHCOM command terminal and the Inspect command terminal.

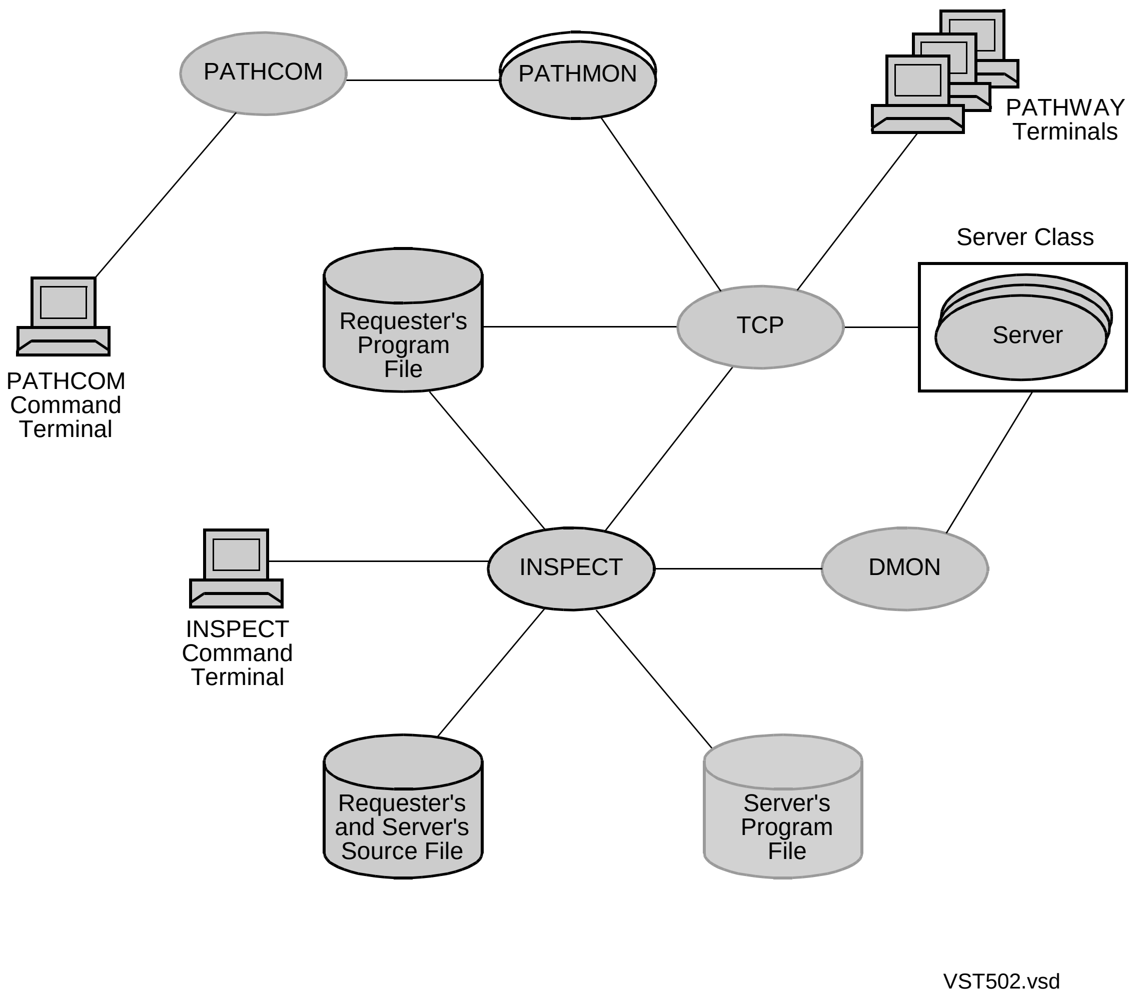
Home terminals are terminals where a process is started. Each process has a home terminal; if a process is started by another process (instead of from a terminal), it inherits the home terminal of its creator. Inspect uses home terminals to determine which terminal it should use as its command terminal.

Debugging PATHWAY Requester Programs

This subsection applies to TNS/R systems, and in some cases to TNS/E systems. Pathway requestors on TNS/E systems (also known as ServerclassSend) can be TNS, TNS/E, or Java, and must be debugged with the appropriate tool. Screen COBOL requestors, however, must be debugged using Inspect. For more information about debugging TNS code accelerated on TNS/E systems, see the *TNSVU Manual* and the *Object Code Accelerator (OCA) Manual*.

Figure 5-2 on page 5-4 highlights the components of Inspect and the PATHWAY system used when debugging requester programs.

Figure 5-2. Debugging PATHWAY Requester Programs



The Inspect process retrieves symbol information from the requester program's program file, and retrieves source information from the requester program's source file or files.

The TCP retrieves pseudocode from the requester program's program file and executes it, controlling one or more PATHWAY application terminals. In addition, the TCP provides the execution control services normally handled by DMON, including:

- Setting and clearing breakpoints
- Providing execution status information
- Retrieving code and data values for display
- Modifying data values

Although PATHCOM and PATHMON are not directly involved in requester program debugging, you use them to configure and prepare the PATHWAY system for requester program debugging. Servers are not directly involved in requester program debugging either, but the data access and retrieval services they provide are required to insure the requester program's proper functioning.

Preparation and Configuration

To use Inspect to debug a requester program, you must first ensure that the requester program's program file includes symbol information. Consequently, you must include the SYMBOLS compiler directive in the source code or on the command line when you compile a SCREEN COBOL program (SCREEN COBOL defaults to NOSYMBOLS).

The symbol information produced by the SYMBOLS directive enables Inspect to associate names of data items with data storage locations, and names of labels and programs with pseudocode locations. If a requester program's program file does not include symbol information, all attempts to debug the requester program with Inspect will produce the error message “Symbol not defined.”

Configuring the TCP

Before you can start debugging a requester program, you must configure its TCP so that the TCP can perform the execution control tasks.

If you are creating a TCP specifically to debug the requester program, you can configure the TCP using the SET TCP INSPECT command provided by PATHCOM:

```
=SET TCP INSPECT ON (FILE $INSPECT-command-term)
```

This SET command performs three functions:

- Configures the TCP so that it can provide execution control services
- Enables the communication link between the TCP and the Inspect process
- Specifies the command terminal on which the Inspect process will run and prompt

Example 5-1 presents an example of configuring and starting a TCP, highlighting the PATHCOM command that enables requester program debugging.

Example 5-1. Starting a TCP for Requester Program Debugging

```
=SET TCP CPUS 8:9
=SET TCP MAXTERMS 5
=SET TCP TCLPROG $pway.reqpgms.pobj
=SET TCP INSPECT ON (FILE $mydbug)
=ADD TCP debug-tcp
=START TCP debug-tcp
```

If you are using an existing TCP, you must configure the TCP using the ALTER TCP Inspect command provided by PATHCOM:

```
=ALTER TCP TCP-name, INSPECT ON (FILE $INSPECT-command-term)
```

Because you can alter the configuration of a TCP only when it is stopped, the actual sequence of commands is:

```
=STOP TCP TCP-name
=ALTER TCP TCP-name, INSPECT ON (FILE $INSPECT-command-term)
=START TCP TCP-name
```

If you do not include the FILE option in the SET or ALTER command, the Inspect process uses the TCP's home terminal as its command terminal.

After you are finished debugging the requester program, you can turn off the TCP's Inspect capabilities using the ALTER TCP Inspect command again:

```
=STOP TCP TCP-name
=ALTER TCP TCP-name, INSPECT OFF
=START TCP TCP-name
```

Note. At one time, a TCP can control a maximum of eight terminals that are involved in Inspect debugging. A TCP can process a total of 20 breakpoints for the set of all requester programs being debugged with Inspect under that TCP.

Starting the Debugging Session

After configuring the TCP for requester program debugging, you can begin debugging requester programs in one of two ways:

- Start the requester program in the Inspect hold state.
- Put a running requester program into the Inspect hold state.

Starting a Requester Program in the Hold State

Starting a requester program in the hold state enables you to put breakpoints in it before the TCP begins executing it. To start a requester program in the hold state, you need to make it the initial program of a PATHWAY terminal, configure that terminal for Inspect debugging, and then start the terminal. The PATHCOM commands SET TERM INITIAL and SET TERM INSPECT enable you to do this:

```
=SET TERM INITIAL requester-program
=SET TERM INSPECT ON (FILE $INSPECT-command-term)
```

Example 5-2 on page 5-7 presents an example of starting a requester program, highlighting the PATHCOM commands that start it in the hold state. Note that the TCP specified in this example is the one configured for requester program debugging in Example 5-2.

Example 5-2. Starting a Requester Program in the Hold State

```
=SET TERM FILE $mypway
=SET TERM INITIAL req-pgm
=SET TERM TCP debug-tcp
=SET TERM INSPECT ON (FILE $mydbug)
=ADD TERM req-term
=START TERM req-term
```

If you do not include the FILE option in the SET TERM INSPECT command, the Inspect process uses the terminal specified in the SET TCP INSPECT command as its command terminal.

Starting to Debug a Running Requester Program

To debug a running requester program, you must force it into the hold state, regardless of how the requester program was started.

To put a running requester program into the hold state, use the PATHCOM INSPECT TERM command:

```
=INSPECT TERM application-term, FILE $INSPECT-command-term
```

This command directs the TCP to put the requester program running on the given application terminal into the hold state, and directs Inspect to use the given command terminal as the Inspect command terminal (provided that the TCP or the terminal has not already specified the command terminal). The TCP puts the requester program into the hold state after it finishes executing the current pseudocode instruction.

If you do not include the FILE option in the INSPECT TERM command, Inspect selects its command terminal as follows:

1. The command terminal specified by the FILE option of the SET TERM INSPECT command for the given application terminal.
2. The command terminal specified by the FILE option of the SET TCP INSPECT command.
3. The TCP's home terminal.

If there is not an Inspect process active on the selected command terminal, $IMON starts one automatically.

If an Inspect process is already active for the selected command terminal, PATHCOM issues a warning and does not change the Inspect command terminal if you specify a different command terminal in the INSPECT TERM command.

Requester Programs Started Using the RUN PROGRAM Command

When you start a requester program using the PATHCOM command RUN PROGRAM, the PATHCOM command terminal usually becomes the application terminal. PATHMON assigns the terminal a synthetic application terminal name.

In this case, you need to determine the synthetic name before you can use the INSPECT TERM command to begin debugging the requester program. Start PATHCOM on another terminal and enter the command:

```
=STATUS TERM *
```

Look for a terminal name of the form nnn-term-mmm, where nnn and mmm are sequence numbers and term is the device name (without the leading dollar sign) of the terminal on which the requester program is running. You use this synthetic name in the INSPECT TERM command to debug the requester program.

Note. When you use RUN PROGRAM to start a requester program, the TCP to which you assign that program (using the SET PROGRAM TCP command) must be configured for Inspect debugging; otherwise, you will not be able to debug the requester program.

Guidelines for Debugging Requester Programs

The following subsections present guidelines that make the task of debugging requester programs easier.

Breakpoints in Requester Programs

- You can set code breakpoints anywhere in an active scope unit of a requester program, and at the entry point of any inactive scope unit.
- Data breakpoints are not permitted in requester programs.
- A TCP can manage a total of twenty breakpoints, regardless of the number of requester programs it is running. For example, if one requester program has ten breakpoints, and two others have five each, you cannot set any more breakpoints in any of the requester programs running under that TCP until you clear one of the existing breakpoints.
- When a requester program reaches a breakpoint, the TCP places the requester program into a debugging hold state and informs the Inspect process, which then performs the break action (if any) associated with the breakpoint. If there is no break action, or if the action did not restore the PATHWAY requester program to the run state, Inspect prompts at its command terminal for a command. When you issue a RESUME or STEP command, Inspect informs the TCP, which then places the PATHWAY requester program back into the run state and resumes executing it.

Using the PATHCOM Terminal as the Inspect Terminal

When debugging requester programs, you should use three terminals: one for the application, one for the PATHCOM command terminal, and one for the Inspect command terminal. You can, however, debug with only two terminals: one for the application and one for both the PATHCOM and Inspect command terminals. Simply specify the PATHCOM command terminal as the parameter to the FILE clause of the INSPECT TERM, SET TERM INSPECT, or SET TCP INSPECT command. After the debugging session begins, you can stop PATHCOM and let Inspect assume complete control of the terminal; follow these steps:

1. Enter the PAUSE command at the Inspect prompt. This command returns terminal control to PATHCOM.
2. Enter the EXIT command at the PATHCOM prompt. This command stops PATHCOM and returns control to the command interpreter.
3. Enter the PAUSE command at the command interpreter prompt. This command pauses the command interpreter and notifies Inspect to start prompting for commands again.
4. You can skip the first step if you are already at the PATHCOM prompt.

Ending the Debugging Session

You can end the debugging session by entering either the EXIT command or the RESUME * EXIT command. Neither of these commands alters the PATHWAY environment, so you should enter PATHCOM and restore PATHWAY to the status it had before you began debugging. This includes such tasks as stopping the requester program and deleting or altering the TCP you configured for Inspect debugging.

When you enter the EXIT command, Inspect terminates the Inspect session, but it leaves the programs that you were debugging in their current states. Consequently, you should use the EXIT command only after you have cleared all breakpoints in all programs and resumed execution of any programs in the hold state. Inspect will prompt you for confirmation if you have any breakpoints set. The RESUME * EXIT command performs these cleanup tasks for you.

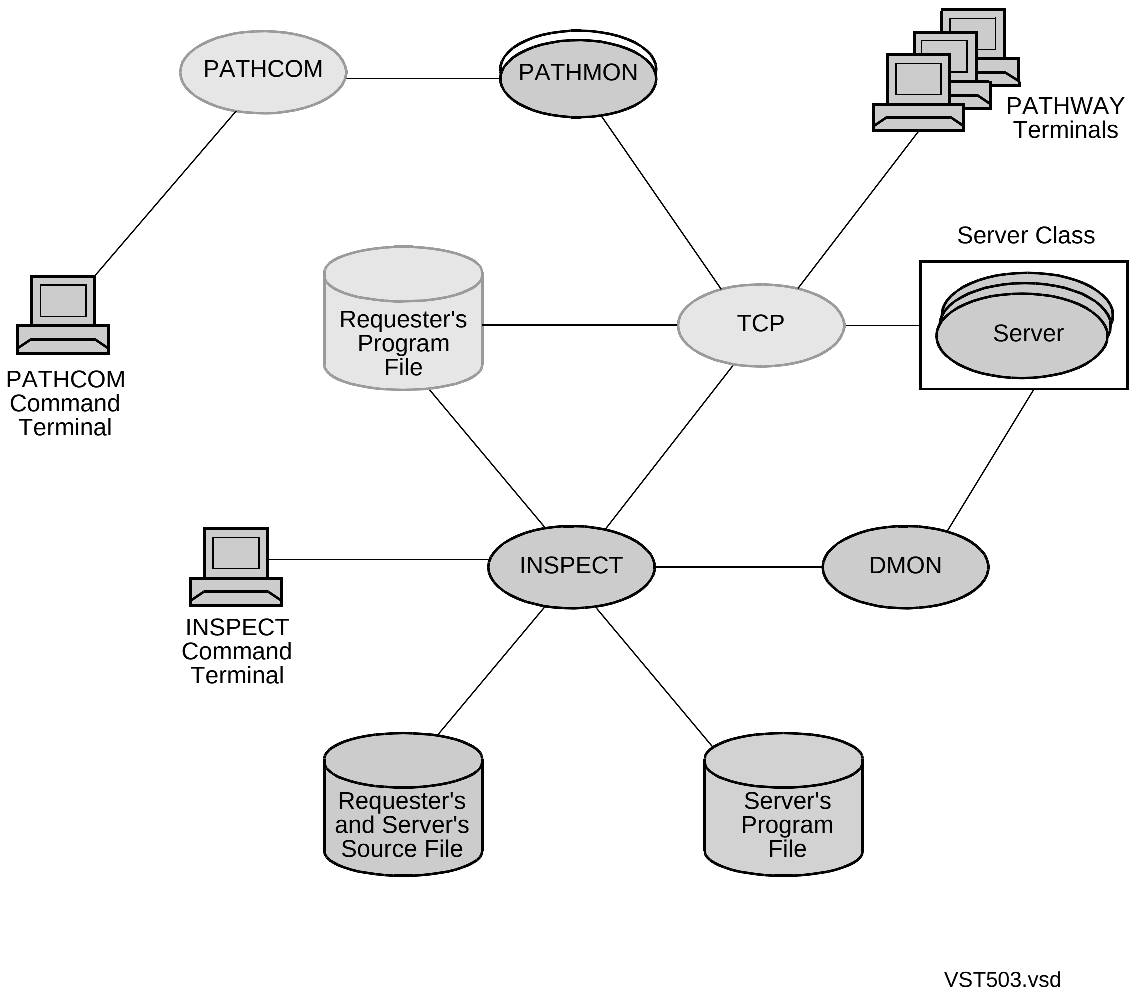
When you enter the RESUME * EXIT command, Inspect clears all breakpoints from all programs on the program list and resumes execution of any programs in the hold state. It then terminates the Inspect session.

If you have paused the command interpreter on the Inspect command terminal, you might have to press the BREAK key to signal the command interpreter that it should prompt again.

Debugging PATHWAY Servers

Figure 5-3 highlights the components of Inspect and the PATHWAY system used when debugging PATHWAY servers.

Figure 5-3. Debugging PATHWAY Servers



Generally, Inspect treats a PATHWAY server like any other process. For example, DMON performs the execution control services for processes and PATHWAY servers, whereas the TCP performs these services for PATHWAY requester programs.

PATHCOM configures servers, PATHMON starts them, and requesters (TCPs running requester programs) control their activity by making data requests. As a result, PATHCOM and PATHMON are involved in configuring a server for debugging, and the server's associated requester must be active to ensure proper execution of the server.

Note that to debug a Pathway server on a TNS/E system, you must use either Visual Inspect or Native Inspect.

Preparation and Configuration

To take full advantage of Inspect, you should ensure that the server's program file includes symbol information. Consequently, you must include the SYMBOLS compiler directive in the source code or on the command line when you compile the server's source code.

Debugging Attributes

As with a process, the debugging attributes of a server and of its creator determine which debugger (Inspect or DEBUG) is used to debug the server. A server's creator is always PATHMON, so you probably cannot control its debugging attributes (unless, of course, you create your own PATHMON specifically for server debugging). As a result, you should compile the server with the Inspect or SAVEABEND debugging attribute on. For more information about debugging attributes and how to set them, see The Debugging Attributes of a Process on page 4-4.

Server Versus Server Class

From an application standpoint, the PATHWAY environment manages server classes, starting and stopping servers in a class as usage demands rise and fall. Inspect, on the other hand, debugs a single server process. You must insure that the server process you are debugging is the same one that PATHWAY will link to your requester. You may want to configure (or reconfigure) the server class so that it represents a single server process, using the PATHCOM commands:

```
=SET SERVER MAXSERVERS 1
=SET SERVER NUMSTATIC 1
```

If the server is running, you can freeze the server class, stop it, reconfigure it, and then restart it:

```
=FREEZE SERVER server-class, WAIT
=STOP SERVER server-class
=ALTER SERVER server-class, MAXSERVERS 1
=ALTER SERVER server-class, NUMSTATIC 1
=START SERVER server-class
```

Starting the Debugging Session

You can start debugging a server in one of two ways:

- Start the server in the Inspect hold state.
- Put a running server into the Inspect hold state.

Starting a Server in the Hold State

Starting a server in the hold state enables you to put breakpoints in it before it begins executing. To start a server in the hold state, you need to set the DEBUG status of its server class before you add the server class. The PATHCOM command SET SERVER DEBUG sets the DEBUG status of a server class:

```
=SET SERVER DEBUG ON
```

Example 5-3 presents an example of starting a server, highlighting the PATHCOM command that starts the server in the hold state. Note that the example includes the configuration commands to limit the server class to a single server and to assign the server's home terminal to the Inspect command terminal.

Example 5-3. Starting a Server in the Hold State

```
.=SET SERVER CPUS 8:9
=SET SERVER PROGRAM $pway.servers.myserver
=SET SERVER MAXSERVERS 1
=SET SERVER NUMSTATIC 1
=SET SERVER HOMETERM $mydbug
=SET SERVER DEBUG ON
=ADD SERVER debug-server
=START SERVER debug-server
```

If you do not use the SET SERVER HOMETERM command, Inspect will use PATHMON's home terminal as its command terminal.

Starting to Debug a Running Server

If you want to debug a running PATHWAY server, you can start to debug it with Inspect by entering the command interpreter DEBUG command:

```
17> DEBUG process-id, TERM $INSPECT-command-term
```

This command activates Inspect, assuming that you have set the debugging attribute of the server (or its controlling PATHMON) to INSPECT ON or SAVEABEND ON. If you do not use the TERM clause, Inspect will use PATHMON's home terminal as the Inspect command terminal.

To discover the *process-id* of the server, use the PATHCOM STATUS SERVER command:

```
=STATUS SERVER server-class, DETAIL
```

Provided that you have limited the server class to a single server (as discussed in Server Versus Server Class on page 5-11.), the STATUS SERVER command will display a single process ID.

After you enter the DEBUG command, the server enters the hold state as soon as it completes executing the current machine-code instruction. If the server has invoked a procedure, it will not enter the hold state until the procedure returns control to the server.

Guidelines for Debugging Servers

Because servers are simply processes, the guidelines for debugging processes apply to servers as well. For more information, see Guidelines for Debugging a Process on page 4-10.

Ending the Debugging Session

You can end the debugging session by entering either the EXIT command or the RESUME * EXIT command. Neither of these commands alters the PATHWAY environment, so you should enter PATHCOM and restore PATHWAY to the status it had before you began debugging. This includes such tasks as stopping and deleting or altering the server.

When you enter the EXIT command, Inspect terminates the Inspect session, but it leaves the programs that you were debugging in their current states. Consequently, you should use the EXIT command only after you have cleared all breakpoints in all programs and resumed execution of any programs in the hold state. The RESUME * EXIT command performs these cleanup tasks for you.

When you enter the RESUME * EXIT command, Inspect clears all breakpoints from all programs on the program list and resumes execution of any programs in the hold state. It then terminates the Inspect session.

If you have paused the command interpreter on the Inspect command terminal, you might have to press the BREAK key to signal the command interpreter that it should prompt again.

Debugging User Conversion Routines

A PATHWAY application can modify the TCP's terminal control logic by adding user conversion routines to the TCP library and then building a new library file for the TCP. For more information about user conversion routines, see the *TS/MP Management Programming Manual.*

Because user conversion routines become part of the TCP's new library file, the TCP must be configured for Inspect before you can debug the user conversion routines that you have written. To ensure that the TCP is configured for Inspect, the Inspect attribute of the TCP's program file, PATHTCP2, is set ON. Consequently, you do not have to take any special configuration steps to debug user conversion routines.

Preparation and Configuration

To take full advantage of Inspect, you should ensure that your user conversion routines include symbol information. Consequently, you must include the SYMBOLS compiler directive in the source code or on the command line when you compile the source code for the user conversion routines.

Starting the Debugging Session

You can start debugging user conversion routines in one of two ways:

- Start the TCP in the Inspect hold state.
- Put a running TCP into the Inspect hold state.

Starting a TCP in the Hold State

Starting a TCP in the hold state enables you to put breakpoints in the user conversion routines before the TCP begins executing. To start a TCP in the hold state, you need to set the TCP's DEBUG status before you add the TCP. The PATHCOM command SET TCP DEBUG sets the DEBUG status of a TCP:

```
=SET TCP DEBUG ON
```

Example 5-4 presents an example of starting a TCP, highlighting the PATHCOM command that configures the TCP to start in the hold state.

Example 5-4. Starting a TCP in the Hold State

```
=SET TCP CPUS 8:9
=SET TCP MAXTERMS 5
=SET TCP TCLPROG $pway.reqpgms.pobj
=SET TCP HOMETERM $mydbug
=SET TCP DEBUG ON
=ADD TCP debug-conv-rtns
=START TCP debug-conv-rtns
```

If you do not use the SET TCP HOMETERM command, Inspect will use PATHMON's home terminal as its command terminal.

If the TCP is running, you can stop it, alter its DEBUG status, and then restart it by entering these commands:

```
=STOP TCP tcp-name
=ALTER TCP tcp-name, DEBUG ON
=START tcp-name
```

This sequence effectively restarts the TCP in the hold state.

Starting to Debug a Running TCP

If you want to debug the user conversion routines in a running TCP, you can start to debug the TCP using one of two methods:

- You can enter the command interpreter command DEBUG:

```
17> DEBUG process-id, TERM $INSPECT-command-term
```

If you do not use the TERM clause, Inspect will use the TCP's home terminal as the Inspect command terminal.

- You can enter the Inspect command ADD PROGRAM:

```
--ADD PROGRAM process-id
```

Both of these methods require that you know the process ID of the TCP. To discover the process ID of the TCP, use the PATHCOM command STATUS TCP:

```
=STATUS TCP tcp-name
```

After you enter the DEBUG or ADD PROGRAM command, the TCP enters the hold state as soon as it completes executing the current machine-code instruction. If the TCP has invoked a procedure, it will not enter the hold state until the procedure returns control to the TCP.

Guidelines for Debugging User Conversion Routines

Because the TCP is a process, the guidelines for debugging processes apply to user conversion routines as well. For more information, see Guidelines for Debugging a Process on page 4-10.

Ending the Debugging Session

You can end the debugging session by entering either the EXIT command or the RESUME * EXIT command. Neither of these commands alters the PATHWAY environment, so you should enter PATHCOM and restore PATHWAY to the status it had before you began debugging.

When you enter the EXIT command, Inspect terminates the Inspect session, but it leaves the programs that you were debugging in their current states. Consequently, you should use the EXIT command only after you have cleared all breakpoints in all programs and resumed execution of any programs in the hold state. The RESUME * EXIT command performs these cleanup tasks for you.

When you enter the RESUME * EXIT command, Inspect clears all breakpoints from all programs on the program list and resumes execution of any programs in the hold state. It then terminates the Inspect session after a prompt.

If you have paused the command interpreter on the Inspect command terminal, you might have to press the BREAK key to signal the command interpreter that it should prompt again.

6 High-Level Inspect Commands

This section describes the high-level Inspect commands in detail. The high-level commands are summarized in Table 6-1 on page 6-2. If you want to make full use of high-level Inspect, your program must be compiled with the SYMBOLS compiler directive or have access to an object with symbols.

Inspect Keywords

Inspect does not reserve its keywords (command names and clause names). Consequently, refer to identifiers such as BREAK, EVERY, and STEP in your program, assuming that the compiler for the language you are using allows them as identifiers.

Abbreviations

You can abbreviate many Inspect keywords. In this section, keyword abbreviations are shown in boldface type. These abbreviations might change to accommodate additional commands in future releases of Inspect, so you should not use the abbreviations in OBEY files, the INSPLOCL file, or the INSPCSTM file.

Language-Dependent Information

Several of the high-level Inspect commands require information that is language-dependent; the syntax and validity of this information is based on the current source language. The command descriptions in this section discuss how the information is used, but you must refer to the particular language sections for details regarding the syntax of the information. The language-dependent information required by a high-level command can be one of these five items:

- Code location—A location in the program's object code area
- Data location—The location of a data item
- Expression—An expression that evaluates to a numeric, boolean (true/false), or string value
- Scope path—A path used to identify specific code and data locations
- Scope unit—The entity (or entities) into which source code is grouped

Machine-Dependent Information

Some high-level Inspect commands may be affected by the type of machine, TNS or TNS/R, on which the current program is executing. The behavior of some commands may be slightly different depending on the machine. Some commands have machine dependent arguments that can only be used on a particular machine, and some commands may report different information depending on the machine. Machine dependencies are noted where appropriate. For more information about TNS/R related features, see Section 15, Using Inspect on a TNS/R System.

Command Examples

This section includes examples that show common usage of the high-level Inspect commands. The examples reflect the output format of the current release of Inspect.

All commands (except the fix commands) have an optional /OUT *file-name*/ clause following the command name.

Command Summary

Table 6-1 lists and describes all the Inspect commands.

Table 6-1. High-Level Inspect Commands (page 1 of 5)

Command	Description
ADD ALIAS	Adds a name to the alias list for the current Inspect session.
ADD KEY	Adds a function-key definition to the function-key list for the current Inspect session.
ADD PROGRAM	Adds a process or save file to the program list for the current Inspect session.
ADD SOURCE ASSIGN	Adds a source assignment to the source assignment list.
ALIAS	Adds a name to the alias list or displays aliases for the current Inspect session.
BREAK	Sets or displays one or more breakpoints in the current program.
CD	Changes the current OSS directory
CLEAR	Clears one or more breakpoints in the current program.
COMMENT (--)	Directs Inspect to ignore the remainder of the command line, therefore enabling you to include remarks in a LOG file, an OBEY file, or the INSPLOCL or INSPCSTM configuration files. A double hyphen can be used instead of COMMENT.
DELETE ALIAS	Removes one or all names from the alias list for the current Inspect session.
DELETE KEY	Removes one or all function-key definitions from the function-key list.
DELETE SOURCE ASSIGN	Removes one or all source assignments from the source assignment list for the current Inspect session.
DELETE SOURCE OPEN	Closes one or all source files that Inspect has opened as the result of previous SOURCE commands.
DISPLAY	Formats and displays a variety of items, including constants, expressions, code, and data.

Table 6-1. High-Level Inspect Commands (page 2 of 5)

Command	Description
ENV	Displays one or all the current settings of the Inspect environment parameters.
EXIT	Stops the Inspect process, therefore terminating the Inspect session.
FA	Enables you to retrieve, edit, and redefine the replacement string for an existing alias.
FB	Enables you to retrieve, edit, and replace the definition of an existing code breakpoint in the current program.
FC	Enables you to retrieve, edit, and execute a command line in the history buffer.
FILES	Shows the status of files that have been opened by the current program.
FK	Enables you to retrieve, edit, and redefine the replacement string for an existing function-key definition.
HELP	Displays information regarding the syntax and usage of Inspect commands and command options.
HISTORY	Displays the most recently executed command lines.
HOLD	Suspends the execution of one or more programs on the program list, placing the program or programs in the hold state.
ICODE	Displays instruction mnemonics starting at the specified code address. For accelerated programs, this command may be used to list TNS instructions, TNS/R instructions, or a combination of both.
IDENTIFIER	Displays information about the internal characteristics of a given data location or of all data locations in one or more scope units. The IDENTIFIER command is a synonym for the INFO IDENTIFIER command.
IF	Provides conditional execution of an Inspect command or an alias.
INFO IDENTIFIER	Displays information about the internal characteristics of a given data location or of all data locations in one or more scope units.
INFO LOCATION	Displays information about a code location in the current program.
INFO OBJECTFILE	Displays information about a current or specified program’s object file.
INFO OPENS	Shows the status of files that have been opened by the current program.
INFO SAVEFILE	Displays information about a current or specified save file.
INFO SCOPE	Displays information about a given scope unit in the current program.

Table 6-1. High-Level Inspect Commands (page 3 of 5)

Command	Description
INFO SEGMENTS	Displays information about the extended segments allocated for or by the current program.
INFO SIGNALS	Displays signal information for the current program.
KEY	Adds a function-key definition or displays one or all function-key definitions in the function-key list for the current Inspect session. The KEY command is a synonym for the ADD KEY and the LIST KEY commands.
LIST ALIAS	Displays one or all aliases from the alias list for the current Inspect session.
LIST BREAKPOINT	Displays one or all breakpoints defined in the current program.
LIST HISTORY	Displays a portion of or the entire history buffer.
LIST KEY	Displays one or all function-key definitions from the function-key list for the current Inspect session.
LIST PROGRAM	Displays the list of programs being debugged.
LIST SOURCE ASSIGN	Displays the source assignments from the source-assignment list for the current Inspect session.
LIST SOURCE OPEN	Displays the names of all source files that Inspect has opened as the result of previous SOURCE commands.
LOG	Records the session input, output, or both input and output on a permanent file.
LOW	Switches Inspect from high-level to low-level command mode.
MATCH	Searches for scope-unit names or other identifiers in the current program.
MODIFY	Changes the value of a data item or register in the current program.
OBEY	Causes Inspect to read commands from a specified file.
OBJECT	Displays information about the current program's object file.
OPENS	Shows the status of files that have been opened by the current program.
OUT	Directs the output listing to a specified file.
PAUSE	Suppresses Inspect prompts until a debug event occurs in any of the programs on the program list.
PROGRAM	Displays the programs in the program list or selects a program as the current program.
RESUME	Reactivates one or more suspended programs, changing the program's state from hold to run.
SAVE	Creates a save file of the current program.
SCOPE	Changes or displays the current scope path.

Table 6-1. High-Level Inspect Commands (page 4 of 5)

Command	Description
SELECT DEBUGGER DEBUG	Invokes Debug on the current program. Once you have used this command, you interact with Debug until issuing the Debug command "INSPECT" to return control of the program to Inspect.
SELECT LANGUAGE	Changes the current source language, changing the acceptable syntax of language-dependent entries.
SELECT PROGRAM	Selects a program from the program list as the current program.
SELECT SEGMENT	Selects extended data segments in which extended data addresses are to be resolved.
SELECT SOURCE SYSTEM	Directs Inspect to retrieve source files from another system when the object file has moved but the source has not.
SELECT SYSTYPE	Changes the current systype of Inspect.
SET	Changes the status of one of the settable Inspect parameters.
SHOW	Shows the status of one or all the settable Inspect parameters.
SIGNALS	Displays signal information for the current program.
SOURCE	Displays source code.
SOURCE ASSIGN	Sets or displays source assignments from the source-assignment list for the current Inspect session.
SOURCE ICODE	Lists instruction mnemonics corresponding to listed source text.
SOURCE OFF	Disables automatic source display at each event.
SOURCE ON	Enables automatic source display at each event.
SOURCE OPEN	Displays the names of the files that are currently open as a result of previous SOURCE commands.
SOURCE SEARCH	Displays source text that matches a specified string.
SOURCE SYSTEM	Directs Inspect to retrieve source files from another system when the object file has moved but the source has not.
STEP	Resumes execution of the current program at the point where it was last suspended, and then suspends execution after the program has executed a certain number of units.
STOP	Stops one or more programs and removes them from the program list.
SYSTEM	Sets the default system for expansion of any file names.
TERM	Changes which terminal or process is the Inspect command terminal.
TIME	Directs Inspect to display the time.

Table 6-1. High-Level Inspect Commands (page 5 of 5)

Command	Description
TRACE	Displays the call history for the current program location.
VOLUME	Sets the default volume and subvolume for expansion of any file names.
XC	Reissues a command line in the history buffer.

ADD

The ADD command adds an item to one of the lists of information that Inspect maintains. This diagram shows the complete syntax for the ADD command and its clauses. Detailed descriptions of the clauses, including usage considerations and examples, are presented in the following subsections.

```
ADD list-item

list-item:   one of

ALIAS alias-name [=] command-string
KEY key-name [=] command-string
PROGRAM program-spec
SOURCE ASSIGN [ original-name ,] new-name

command-string:   one of

" [ character ]... "
' [ character ]... '

key-name:   one of

F1     F2     F3     F4     F5     F6     F7     F8
F9     F10    F11    F12    F13    F14    F15    F16
SF1    SF2    SF3    SF4    SF5    SF6    SF7    SF8
SF9    SF10   SF11   SF12   SF13   SF14   SF15   SF16

program-spec:   one of

process
save-file [ CODE code-file ]
          [ LIB lib-file ]
          [ SRL {(srl-file [ , srl-file,...])}]
```

```
original-name:   one of

[ \system. ] $volume [ .subvolume [ .file ] ]
[ \system. ] $process [ .#qual-1 [ .qual-2 ] ]
[ \system. ] cpu, pin
[ \system. ] $volume.#number
/oss-pathname [/oss-pathname... ]

new-name:

[\system.] $volume [ .subvolume [ .file ] ]
/oss-pathname [/oss-pathname... ]
```

Related Commands

- DELETE on page 6-29
- LIST on page 6-129
- SELECT on page 6-164

ADD ALIAS

The ADD ALIAS command adds a name for a command string to the alias list for the current Inspect session. To add an alias, you provide ADD ALIAS with:

- The name of the alias
- The command string that the alias name represents

After you add an alias, you can use its name whenever you would normally enter its replacement string.

```
ADD ALIAS alias-name [=] replacement-string


replacement-string:   one of

" [ character ]... "
' [ character  ]... '
```

alias-name

specifies the name of the alias. This name can contain up to 31 alphanumeric characters; the first character must be alphabetic. An alias name cannot be the same as an Inspect command name or command name abbreviation.

replacement-string

specifies the command string to associate with the alias name. The command string is a group of zero or more characters enclosed in either quotes (") or apostrophes ('). To include a quote in a quote-delimited replacement string, use a pair of quotes. Likewise, to include an apostrophe in an apostrophe-delimited replacement string, use a pair of apostrophes.

Usage Considerations

- Restrictions on the Contents of the Replacement String

  An alias replacement string can refer to other aliases (by name), but it cannot contain the XC command or any of the Fix commands (FA, FB, FC, and FK).

- Alias Restrictions

  - Aliases are not expanded for a pattern in the MATCH IDENTIFIER or the MATCH SCOPE commands.
  - Aliases are not expanded in HELP command parameters or with the HELP command.
  - Aliases are not expanded following the keyword ALIAS in the ALIAS, ADD ALIAS, DELETE ALIAS, and LIST ALIAS commands.

- Using Aliases in the BREAK THEN Clause

  You can use an alias name as the parameter to the THEN clause of the BREAK command; for example:

```
ADD ALIAS ShowMe = "DISPLAY pitch,roll,yaw,speed;RESUME"
BREAK #inject.verify THEN ShowMe
```

  When the break event occurs, Inspect executes the command list specified in the alias replacement string. If the command list includes a RESUME command, Inspect does not prompt for a command; instead, it resumes execution immediately.

- Recursive Problems with Aliases

  This examples illustrate how attempting to add a recursive alias will cause direct recursion, as in:

```
ADD ALIAS MYPROC = "MYPROC"
```

  will cause this error:

```
** Inspect error 380 ** Recursive alias definition
```

  Indirect recursion, as in:

```
ADD ALIAS MYPROC = "MYPROC;DISPLAY MYDATA;RESUME"
```

will result in this error message being displayed:

```
** Inspect error 4 ** Effective input record is too long
```

Related Commands

- ALIAS on page 6-17
- DELETE ALIAS on page 6-30
- FA on page 6-83
- LIST ALIAS on page 6-131
- SET ECHO with the ALIAS option on page 6-174

Examples

Using the IF command and references to other aliases within an alias:

```
-PRG-ADD ALIAS oldval = "IF x < 99 THEN DISPLAY 'X hack ',X"
-PRG-ADD ALIAS xset = "DISPLAY 'X being set to 99';MOD X=99"
-PRG-ADD ALIAS xhack = "oldval;xset"
```

Entering XHACK causes expansion of OLDVAL and XSET.

```
-PRG-xhack
x hack x=50
x being set to 99
-PRG-DISPLAY X
x=99
```

Using aliases as abbreviations for long scope or identifier names:

```
-PRG-ADD ALIAS myproc = “A_Very_Long_Procedure_Name”
-PRG-BREAK #myproc
```

ADD KEY

The ADD KEY command adds a function-key definition to the function-key list for the current Inspect session. To add a function-key definition, you provide ADD KEY with:

- The name of the function key

- The command string that the function key will execute

```
ADD KEY key-name [=] command-string



key-name:  one of
F1    F2    F3    F4    F5    F6    F7    F8
F9    F10   F11   F12   F13   F14   F15   F16
SF1   SF2   SF3   SF4   SF5   SF6   SF7   SF8
SF9   SF10  SF11  SF12  SF13  SF14  SF15  SF16


command-string:  one of
" [ character ]... "
' [ character ]... '
```

key-name

specifies the function key for which you want to provide a definition. Valid function keys include F1 through F16 and shifted F1 (SF1) through shifted F16 (SF16).

command-string

specifies the replacement string to associate with the given function key. The replacement string is a group of zero or more characters enclosed in either quotes (") or apostrophes ('). To include a quote in a quote-delimited replacement string, use a pair of quotes. Likewise, to include an apostrophe in an apostrophe-delimited replacement string, use a pair of apostrophes.

Usage Consideration

A function key's replacement string cannot contain the XC command or any of the Fix commands (FA, FB, FC and FK).

Related Commands

- DELETE KEY on page 6-30
- FK on page 6-90
- LIST KEY on page 6-136
- SET ECHO on page 6-174

ADD PROGRAM

The ADD PROGRAM command adds a process or save file to the program list for the current Inspect session. This command is used to place a process under Inspect’s control (instead of using the TACL RUND command) or to examine the contents of a save file.

```
ADD PROGRAM { process                                }
{ save-file [ CODE code-file]
            [ LIB lib-file] ]
            [ SRL {(srl-file [ , srl-file,...])}]

process:

  [\system-name.] { $name | cpu,pin | oss-pid }
```

process

specifies a process as the program to add. $*name* identifies the process by its process name, *cpu,pin* identifies it by its process ID (CPU number and process number), and *oss-pid* identifies an OSS process by its OSS process ID.

```
{ save-file [ CODE code-file [ LIB lib-file] ]
             [ SRL {(srl-file [ , srl-file,...])}]
```

specifies a save file as the program to add. Inspect retrieves the file, adds it to the program list, and makes it the current program. Depending on Inspect's current systype, this can either be a Guardian file name, or an OSS pathname.

The systype determines the syntax of the file name, not the type of the process being added. The CODE and LIB clauses enable you to associate alternate code and library files with the given save file. The SRL clause is a single filename, or a comma separated list of SRL object files within parentheses. If the process used more SRLs than you specified, Inspect will emit a warning.

CODE *code-file*

directs Inspect to retrieve symbol information from an object code file other than the one specified internally in the save file.

LIB *lib-file*

directs Inspect to retrieve symbol information from a library file other than the one specified internally in the save file.

SRL *srl-file*

directs Inspect to retrieve symbol information from a shared runtime library file other than the one specified internally in the save file.

Note. The SELECT PROGRAM command can be used to specify alternate code and library files with a running process after it has been successfully added.

Usage Considerations

- PATHWAY Requester Programs

  To add a PATHWAY requester program to the Inspect program list, you must use the PATHCOM command INSPECT TERM.

- Cautions When Adding Processes

  When you use ADD PROGRAM to add a process, the process will not be added to the current Inspect session in these cases:

  - If the Inspect attribute of the process is not set. In this case, the operating system starts a Debug session for the process.
  - If the process is executing on a remote system. In this case, IMON starts an Inspect process (and, consequently, a new Inspect session) on the remote process's host system.
  - If your user ID does not have permission to control the process.

  After all these checks pass, the process will be added to the Inspect program list after the process exits system code.

- Using the CODE and LIB Clauses

  The CODE and LIB clauses enable you to associate save files with object and library files other than those specified internally in the save file. Inspect provides these clauses so that you can obtain symbol information even when the object or library file doesn't contain symbols. For example, most applications include symbols only during their development; the symbol information is stripped out before distribution. If a bug is then discovered, the customer can make a save file and return it to the developers. Using the CODE and LIB clauses, the developers can then associate their versions of the object and libraries (with symbols) to the save file, therefore enabling them to use high-level, symbolic Inspect to pinpoint the problem more quickly.

  These clauses are also useful when the code or library files are no longer stored at the location saved in the save file.

- Save Files Created when a Process ABENDs

  When a process, which has its SAVEABEND attribute set, terminates abnormally, DMON automatically creates a save file in the volume and subvolume containing the program file. The save file that DMON creates has a file code of 130 and a name of the form ZZSAdddd, where dddd is a number chosen by DMON.

  When DMON creates the save file, it prints this message at the home terminal of the abending process:

```
Savefile File Created:  file-name
```

  If the home terminal is not available or is busy, DMON does not print the message.

The ZZSA file is created in the same subvolume the program is running from. This could be different from your logon subvolume. Enter this command at the TACL prompt:

```
5>FILEINFO zzsa*
```

Then look for the save file whose modification timestamp matches the time of the abnormal termination.

- Save Files and Timestamps

  When Inspect retrieves a save file for analysis, you can receive warning messages giving timestamp information.

  Each object file in the system contains a BINDER timestamp. When the Inspect command SAVE creates a save file, it includes this timestamp. If you use the ADD PROGRAM command to fetch a save file and the recorded timestamp in the save file does not match the timestamp in the corresponding program file on disk, you receive a warning message because the program file might have been modified.

- Adding Programs on a TNS/E System

  You cannot add a TNS/E process or a TNS/E snapshot file to an Inspect session on a TNS/E system. Instead, you must use either Visual Inspect or Native Inspect to debug a TNS/E process or snapshot.

Related Commands

- INFO SAVEFILE on page 6-120
- LIST PROGRAM on page 6-137
- PROGRAM on page 6-156
- SELECT PROGRAM on page 6-167
- STOP on page 6-215

Examples

1. This example assumes there is a previously created save file for $NAMEX.

```
--ADD PROGRAM zzsa1630
-$NAMEX-
...
-$NAMEX-STOP
--COMMENT  STOP closes the save file without terminating INSPECT
--EXIT
```

2. In this example, a new object file is associated with a save file. The new object file might have been compiled with the SYMBOLS compiler directive; the old one was not.

```
--ADD PROGRAM zzsa5362 CODE testprog
```

or

```
--ADD PROGRAM zzsa4513 LIB sort
```

ADD SOURCE ASSIGN

The ADD SOURCE ASSIGN command adds a source assignment to the current program's source assignment list.

When a program is compiled, the fully qualified names of the source files that compose it are recorded as part of the symbol information. Inspect uses this information to determine what file to retrieve source text from. If a source file has been moved since a program was compiled, Inspect will be unable to locate source text to display. The ADD SOURCE ASSIGN command enables you to inform Inspect where to find source files when their location has changed.

```
ADD SOURCE ASSIGN [ original-name , ] new-name


original-name:   one of

   [ \system. ] $volume [ .subvolume [ .file ] ]
   [ \system. ] $process [ .#qual-1 [ .qual-2 ] ]
   [ \system. ] cpu, pin
   [ \system. ] $volume.#number
   /oss-pathname [/oss-pathname... ]

new-name:   one of

   [\system.] $volume [ .subvolume [ .file ] ]
   /oss-pathname [/oss-pathname... ]
```

original-name

specifies the name of a volume, subvolume, file (permanent or temporary), or process where Inspect would normally look for source code. Note that the volume name is required for a permanent or temporary file. If omitted, Inspect uses the file of the current scope. Inspect does not allow you to omit the *original-name* when debugging SCOBOL programs.

new-name

specifies the name of the volume, subvolume, or file where you want Inspect to look for source code when it would normally look in *original-name*. Note that *volume* is required.

new-name must be qualified down to the same level as *original-name*. That is, if *original-name* is a volume, *new-name* must be a volume; if *original-name* is a subvolume, *new-name* must be a subvolume; if *original-name* is a file or process, *new-name* must be a file. If *original-name* is omitted, *new-name* must be fully qualified.

Usage Considerations

- How Inspect Applies Source Assignments

After Inspect retrieves a compilation name, it scans the source assignment list for original names that match the compilation name. Inspect matches names from left to right, and tests for both partial and complete matches. If, for example, the compilation name is $vol.svol.file, Inspect matches it with these source assignments:

```
ADD SOURCE ASSIGN $vol,           $vol1
ADD SOURCE ASSIGN $vol.svol,      $vol2.subvol2
ADD SOURCE ASSIGN $vol.svol.file, $vol3.subvol3.file3
```

When more than one original name matches a source assignment, Inspect chooses the one that matches the source file to the greatest length. Using the preceding assignments, this table lists the selection and effect of various assignments:

Compilation Name	Assignment Applied	Resulting Current Name
$vol.test.file	first	$vol.test.file
$vol.svol.oldfile	second	$vol2.subvol2.oldfile
$vol.svol.file	third	$vol3.subvol3.file3
$vol.svol.filename	second	$vol2.subvol2.filename

The final example in this table demonstrates that Inspect does not match on a character-by-character basis; instead, it matches on a name-by-name basis. The original name of the third assignment ($vol.svol.file) is a character match of the source assignment $vol.svol.filename, but Inspect does not use it because the two filenames (“file” and “filename”) do not match. Instead, Inspect uses the second assignment because its original name ($vol.svol) is an exact name match of the original name down to the subvolume level.

- Name Qualification of Compilation Names

Compilation names in the symbol region are qualified up to the system level. As a result, all source assign references to such compilation names must include the system name as well. For example, if the compilation name is

\sys1.$devel.work.v1, and if this source file has been moved to \archive.$source, the assignment is:

```
ADD SOURCE ASSIGN \sys1.$devel, \archive.$source
```

If `\sys1` cannot be seen from the system Inspect is running on, you need to use the ADD SOURCE ASSIGN command without specifying the original-name. Alternatively, you can omit the system-name in the original-name. For example:

```
-PROGRAM-ADD SOURCE ASSIGN $OLDVOL.OLDSUBV, \SYS2.$NEWVOL.NEWSUBV
Created:  Original $OLDVOL.OLDSUBV  Current \SYS2.$NEWVOL.NEWSUBV
```

- Restrictions on the New Name

  The new name you provide must refer to a permanent disk file; it cannot refer to a process or a temporary file.

- Location of Source Files

  If you used a C-series compiler and your object file has been moved to another node on a network and the source files are at the same location as when the object file was compiled, use the SELECT SOURCE SYSTEM command to identify the node that the files reside on.

- OSS Pathnames as Source Assigns

  OSS pathnames are accepted as source assigns. In this case, filenames are not dependent on the current systype of Inspect. All OSS pathnames must begin with a “/” character.

- Source Assigns from an OSS file to an Guardian EDIT File

  When adding a source assign from an OSS file to a Guardian EDIT file, or vice versa, unless the EDIT file line numbers match the sequential line ordinal of the ASCII file, Inspect may not display the correct source.

- Compilation Filename Matching

  Inspect requires complete filenames for a source assign to match. For example, the command ADD SOURCE ASSIGN /usr/s, /usr/tmp will not match with a compilation filename of /usr/src/file.c, but will match against /usr/s/file.c. The resulting source file will be usr/tmp/file.c. In other words, Inspect will match on a name-by-name basis, not a character-by-character basis.

- Guardian Compilation Filenames

  Inspect will only compare Guardian compilation file names against Guardian original-names, and vice versa. The new-name can be from a different file system than the compilation name. For example:

```
-PROG-ADD SOURCE ASSIGN /usr/src/file.c,$vol.subvol.filec
Created:  Original /usr/src/file.c  Current $VOL.SUBVOL.FILEC
-PROG-ADD SOURCE ASSIGN $vol.subvol.module, /usr/src/module.c
Created:  Original $vol.subvol.module  Current /usr/src/module.c
```

Related Commands

- DELETE SOURCE ASSIGN on page 6-31
- LIST SOURCE ASSIGN on page 6-141
- SELECT SOURCE SYSTEM on page 6-169
- SOURCE on page 6-196
- SOURCE ASSIGN on page 6-202
- SOURCE SYSTEM on page 6-211

Examples

1. If the source files have been moved to another volume, but have the same subvolume name, issue a command of this form:

```
ADD SOURCE ASSIGN $oldvol, $newvol
```

2. If the source files have been moved to another volume and subvolume, issue a command of this form:

```
ADD SOURCE ASSIGN $oldvol.oldsubv, $newvol.newsubv
```

3. If the location of a single source file has changed, issue a command of this form:

```
ADD SOURCE ASSIGN $oldvol.oldsubv.oldfile, $newvol.newsubv.newfile
```

ALIAS

The ALIAS command adds a name or replacement string or displays aliases and replacement strings for the current Inspect session. The ALIAS command is a synonym for the ADD ALIAS and LIST ALIAS commands. To add an alias, you provide ALIAS with:

- The name of the alias
- The replacement string that the alias name represents

After you add an alias, you can use its name whenever you would normally enter its replacement string. To display an alias, provide the name only.

```
ALIAS[ES] [ alias-name [ [ = ] replacement string ] ]

replacement-string:   one of

" [ character ]..." ]
' [ character ]...' ]
```

`alias-name`

specifies the name of the alias. This name can contain up to 31 alphanumeric characters; the first character must be alphabetic. An alias name cannot be the same as an Inspect command name or a command name abbreviation.

`replacement-string`

specifies the replacement string to associate with the alias name. The replacement string is a group of zero or more characters enclosed in either quotation marks (") or apostrophes ('). To include a quote in a quote-delimited replacement string, use a pair of quotes. Likewise, to include an apostrophe in an apostrophe-delimited replacement string, use a pair of apostrophes.

Default Values

- If you do not specify an *`alias-name`* or a *`replacement-string`*, ALIAS displays all alias definitions.
- If you specify an *`alias-name`* but not a *`replacement-string`*, ALIAS displays the alias definition for *`alias-name`*.

Usage Considerations

- If you specify *`alias-name`* and *`replacement-string`*, ALIAS will add an alias definition for *`alias-name`*.
- Aliases are not allowed with the MATCH command.

Related Commands

- ADD ALIAS on page 6-7
- DELETE ALIAS on page 6-30
- FA on page 6-83
- LIST ALIAS on page 6-131

BREAK

The BREAK command allows you to display all breakpoints or set one or more breakpoints in the current program. The BREAK command, when used without any parameters, is a synonym for the LIST BREAKPOINT command.

```
BREAK [ breakpoint [ , breakpoint ]...]


breakpoint:

   brk-location [ brk-condition ]... [ brk-action ]...


brk-location:

   { code-location    } [ BACKUP ]
   { data-location  [ data-subtype ] [ BACKUP ] }
   { [#] ABEND        }
   { [#] STOP         }


data-subtype:   one of
   ACCESS
   CHANGE
   READ
   WRITE
   READ WRITE
   WRITE READ


brk-condition:   one of
   EVERY integer
   IF expression


brk-action:   one of
   TEMP [ integer ]
   THEN { command-string | alias-name }
```

breakpoint

defines the attributes of a breakpoint. These attributes consist of:

- Location—where the break event is to occur
- Conditions—under what conditions the break event is to occur
- Actions—what to do after the break event occurs

brk-location

specifies the location of the breakpoint.

`code-location`

specifies the location of a code breakpoint. It must be the location of an executable instruction in the user code or user library code.

For PATHWAY requester programs, code-location must be in an active scope unit or at the start of an inactive scope unit.

`BACKUP`

specifies that the breakpoint refers to the backup process of a fault-tolerant process pair.

The BACKUP clause is invalid for PATHWAY requester programs and is not supported on TNS/E systems.

`data-location`

specifies the location of a data breakpoint. If data-location specifies a data item that occupies more than one word (16 bits) of memory, Inspect sets the data breakpoint at the first word of the data item.

The *`data-location`* parameter is invalid for PATHWAY requester programs.

`data-subtype`

specifies where the break event should occur.

`ACCESS`

specifies that a break event should occur on both read and write access of the data item. This keyword can be used as a substitute for the keywords READ WRITE or WRITE READ.

`CHANGE`

specifies that a break event should occur only when the value of the data item changes. Change is the default.

`READ`

specifies that a break event should occur on both read and write access of the data item.

`WRITE`

specifies that a break event should occur on write access of the data item.

`READ WRITE`

specifies that a break event should occur on both read and write access of the data item.

`WRITE READ`

specifies that a break event should occur on both read and write access of the data item.

`ABEND`

specifies a break on ABEND is a break on the event for the process being debugged, not the system procedure.

The ABEND clause is invalid for PATHWAY requester programs.

`STOP`

specifies a break on STOP is a break on the event for the process being debugged, not the system procedure.

The STOP clause is invalid for PATHWAY requester programs.

`brk-condition`

specifies whether encountering the break location should trigger a break event. Inspect provides two *`brk-condition`* clauses, EVERY and IF; you can use these clauses separately or together. When they are used together, the EVERY clause is evaluated before the IF clause.

`EVERY` *`integer`*

specifies that the breakpoint should trigger a break event every integer times the break location is encountered.

The maximum value of *`integer`* is 32767.

`IF` *`expression`*

specifies that the breakpoint should trigger a break event only if *`expression`* evaluates to TRUE (that is, a nonzero value). Because Inspect performs this test when the break location is encountered, all unqualified variable references in *`expression`* inherit the scope path of the scope unit containing the break location. Syntax and semantic checking are not done until the breakpoint has been triggered.

`brk-action`

specifies what actions are to occur automatically when the breakpoint generates a break event. Inspect provides two *`brk-action`* clauses, TEMP and THEN; you can use these clauses separately or together.

`TEMP [` *`integer`* `]`

directs Inspect to clear the breakpoint after it triggers a break event *`integer`* times. If you omit integer, Inspect assumes the *`integer`* value 1.

`THEN {` *`command-string`* `|` *`alias-name`* `}`

directs Inspect to execute a list of commands when the breakpoint triggers a break event. This list of commands is defined either by an explicit command string or by an alias name.

`command-string`

is an Inspect command list enclosed in either quotation marks (") or apostrophes (').

`alias-name`

is the name of an alias whose replacement text is a command list. The alias need not exist when you define the breakpoint; however, it must exist when the breakpoint is activated. Syntax and semantic checking are not done until the breakpoint has been triggered.

General Usage Considerations

- Listing All Breakpoints

  Use the BREAK command without any parameters to list all breakpoints in your current program.

- Breakpoints at Primary Entry Points

  A break event at the primary entry point of a scope unit suspends execution before the scope unit allocates and initializes its local data. In general, you can perform the initialization by entering a STEP 1 STATEMENT command after the break event.

- Breakpoints Limits

  The numbered breakpoint limit in a program is 99. After you have exceeded 99 breakpoints, Inspect maintains the breakpoint list, but no longer numbers them. For more numbered breakpoints, delete unnecessary breakpoints. The breakpoint limit in a TCP is 20 per TCP.

- Changing the Conditions and Actions of a Breakpoint

  You can alter a breakpoint's conditions and actions by entering a BREAK command with the same break location but different break conditions and break actions. No prior CLEAR is needed. Inspect will report that the old breakpoint was replaced.

- Breakpoint Definition versus Breakpoint Activation

  Inspect processes some of a breakpoint's attributes when you define the breakpoint (breakpoint definition), and others when the breakpoint is encountered (breakpoint activation). Because the scope path and input radix used for qualification might change between definition and activation, you should exercise caution when using unqualified identifiers or numbers in a breakpoint definition.

This item is qualified using the scope path current at breakpoint definition:

- Identifiers used to specify the break location

These items are qualified using the input radix and the scope path current at breakpoint activation:

- Numbers appearing in the expression of the IF clause
- Identifiers appearing in the expression of the IF clause
- Identifiers appearing in the command string of the THEN clause

- Debugging Loops

  When debugging loops, you can use the EVERY clause to break only on certain iterations. For example, suppose a loop malfunctions on the 32nd iteration (or every 32nd iteration). If you set an unconditional breakpoint in the loop, it will stop execution 31 times before getting to the point at which the loop malfunctions. If you add EVERY 32 to that unconditional breakpoint, execution won't be stopped until it gets to the point you want.

- Testing EVERY and IF Clauses

  If you use EVERY and IF together, Inspect tests the EVERY clause before testing the IF clause. That is, Inspect tests the IF clause only if the EVERY test passes. For example, Inspect tests the IF clause of this breakpoint only on every 42nd activation:

  ```
  BREAK para-1 EVERY 42 IF c < 100
  ```

  The condition of this breakpoint is “on every 42nd iteration break if C is less than 100” instead of “break every 42nd time that C is less than 100.”

- Debugging a Backup Process

  When debugging a fault-tolerant process pair on a TNS/R system, remember that Inspect puts breakpoints in the primary process unless you use the BACKUP clause. After you have put breakpoints in the backup process, you can stop the primary process (using either the command interpreter STOP command or the INSPECT STOP command) to force control to pass to the backup process.

  If a fault-tolerant process pair alters its home terminal, activation of a breakpoint in the backup process might cause IMON to start a new Inspect process.

  Note that you cannot debug a backup process on a TNS/E system; the BACKUP option is disabled because the backup process is not owned by the debugger on a TNS/E system.

- Using the TEMP Clause

  A breakpoint normally remains in effect until you explicitly clear it using the CLEAR command. Frequently, however, you might want a breakpoint that triggers a break event a limited number of times (usually once), and then clears itself automatically. The TEMP clause provides this feature.

For example, this breakpoint generates break events on its 10th, 20th, and 30th activations, and is then cleared:

```
BREAK para-1 EVERY 10 TEMP 3
```

- Using the THEN Clause

  The THEN clause, which requires quotes, enables you to execute a list of commands whenever a break event is triggered. If you use RESUME in the command list (it must be the last command in the list), Inspect resumes execution of the current program. This stop-and-go feature has a variety of uses, including:

  - Resetting the value of a variable:

    ```
    BREAK #loop.init THEN "MODIFY counter=1;RESUME"
    ```

  - Tracking the value of a variable:

    ```
    BREAK #main.status THEN "DISPLAY status;RESUME"
    ```

  - Flagging execution of a certain piece of code:

    ```
    BREAK #parse THEN "DISPLAY 'Parsing input';RESUME"
    ```

  - Setting a local data breakpoint, which must be done when the associated procedure is active:

    ```
    BREAK #loop.init THEN "BREAK counter; RESUME"
    ```

  If you STEP onto a breakpoint with a RESUME in the THEN clause, your program resumes.

- Using OBEY in a THEN Clause

  Using an OBEY command in a THEN clause can cause unexpected synchronization behavior. For more information, see OBEY on page 6-152.

Usage Considerations for Data Breakpoints

- Restrictions on Data Breakpoints

  Inspect does not allow data breakpoints in PATHWAY requester programs. In addition, Inspect allows only one data breakpoint at a time in each process or PATHWAY server.

- Change is the Default

  If the type of data access that triggers the data breakpoint event is not specified, the default is change. When data breakpoint is set to change, only memory writes that change the value of the data location triggers the breakpoint.

  The READ clause specifies that a break event should occur on both read and write access of the data item.

- Write Availability

  Write breakpoints are available on all TNS processors, but not on NSR-L processors. When a high-level write breakpoint is reported (TNS systems only) and the value of the variable has not changed, Inspect issues this warning:

```
** Inspect warning 364 ** Value of variable did not change;
                          breakpoint may have been triggered by an access
                          to the containing 16-bit word
```

- Machine Dependency

  The program location after a data breakpoint is triggered is machine-dependent. On TNS systems, it might differ by an instruction. On TNS/R systems, it might differ by more than an instruction. For more information, see Section 15, Using Inspect on a TNS/R System.

- Single-Byte Data Objects

  A data breakpoint is associated with a single 16-bit word; therefore, if you set a breakpoint on a data object that occupies a single byte, the breakpoint is associated with the word that contains the data object. Subsequent access of either byte of the word generates a break event for read and write breakpoints.

- Multi-Word Data Objects

  A data breakpoint is associated with a single 16-bit word; therefore, if you set a breakpoint on a data object that occupies multiple words (a floating point value, or character field in record or structure, for example), the breakpoint is associated with the first word of the data object. If your program accesses one of the other words of the data object without affecting the first word, a break event does not occur.

- Byte Data starting on an Odd Byte

  A field in a record or structure might start on an odd byte boundary. Setting a write or read/write data breakpoint for the field might result in unrelated breakpoints being reported when the trailing byte of the previous field is accessed.

- Initialization

  Data breakpoints should be set after initialization has taken place. Variable locations might not be established in memory until initialization is complete.

- Local and Sublocal Data Objects

  A data breakpoint set on a local or a sublocal variable persists even after its containing scope unit has completed execution. It persists because it is associated with a given physical address. Consequently, if a scope unit later allocates a local or sublocal variable at the same physical address, that new variable has the old data breakpoint associated with it.

This illustrates how to set a data breakpoint on a local variable to avoid unrelated data breakpoints being reported for the local variable when the procedure is no longer active.

```
#10   PROC X;
#11   BEGIN
#12      INT I;
#13
#14      CALL Y(i);
         .
         .
         .
#42   END;

-PROGRAM-BREAK #X THEN "BREAK I;RESUME"
-PROGRAM-BREAK #X.#42 THEN "CLEAR I;RESUME"
```

- TAL P-Relative Arrays

  Because TAL P-relative arrays are stored in the code space rather than the data space, you must use the READ clause to set a data breakpoint at a P-relative array.

Usage Consideration for Code Breakpoints

- STOP and ABEND for Code Breakpoints

  If you have a data item or a code label named STOP visible from the current scope unit, the command BREAK STOP sets a breakpoint on the data item or the code label. To set a breakpoint that is reported when the program stops, preface the word STOP with a “#”, as in BREAK #STOP.

  Conversely, if you have a scope unit in your program named STOP, and you issue the command BREAK #STOP, a breakpoint is placed in your program. If you have both a data item or a code label named STOP and a scope unit named STOP, you cannot set a breakpoint at STOP. The same alternatives and restrictions apply for setting a breakpoint at ABEND. To avoid restrictions on the use of STOP and ABEND breakpoints, refrain from declaring data items, labels, or scopes named STOP or ABEND.

Related Commands

- CLEAR on page 6-27
- FB on page 6-84
- IF on page 6-103
- LIST BREAKPOINT on page 6-131

Example

This example illustrates a frequent use of the THEN clause—modifying a variable and then continuing execution:

```
-PRG-BREAK rachets+3I THEN "MODIFY fsize:=15; RESUME"
```

CD

The CD command changes the current OSS directory.

```
CD [ oss-pathname ]
```

oss-pathname

specifies an OSS pathname.

Default Value

If the *oss-pathname* is omitted, the current OSS directory is changed to your initial OSS directory.

CLEAR

The CLEAR command clears one or more breakpoints in the current program.

```
CLEAR { * | clear-spec }


clear-spec:   one of

   breakpoint-number [, breakpoint-number ]
   CODE code-location-list [ , code-location ]
   DATA data-location [ , data-location ]
   EVENT { ABEND | STOP }
```

*

clears all breakpoints in the current program.

clear-spec

identifies the breakpoint to clear. This breakpoint identifier can be the number of the breakpoint (as shown by the LIST BREAKPOINT command) or the location of the breakpoint (a code location, a data location, the keyword STOP, or the keyword ABEND).

breakpoint-number

specifies the number of the breakpoint to clear.

code-location

specifies the location of a code breakpoint. It must be the location of an executable instruction in the user code or user library code.

data-location

specifies the location of a data breakpoint.

Related Commands

- BREAK on page 6-19
- LIST BREAKPOINT on page 6-131

COMMENT

The COMMENT command directs Inspect to ignore the remainder of the command list. It is a way to document commands in an OBEY file or the INSPLOCL or INSPCSTM configuration files. If you are logging input, the file will contain any comments entered.

```
COMMENT | -- [ text ]
```

text

is any text.

Usage Consideration

There are two forms of the COMMENT command in Inspect; COMMENT and “--”. COMMENT must appear at the beginning of a command to be recognized. “--” may appear within a command; subsequent text is then ignored.

Related Command

- LOG

Examples

1. This is an example of a valid comment using COMMENT.

```
-PROG-BREAK #PROC.#42;COMMENT Set a breakpoint in #PROC
```

2. This is an example of an invalid comment; Inspect will issue an invalid syntax error.

```
-PROG-BREAK X COMMENT BREAK when X changes.
```

3. This is an example of a comment using “--”.

```
-PROG-BREAK X -- BREAK when X changes.
```

DELETE

The DELETE command removes an item from one of the lists of information that Inspect maintains.

This diagram shows the complete syntax for the DELETE command and its clauses. Detailed descriptions of the clauses, including usage considerations and examples, are presented in the following subsections.

```
DELETE list-item


list-item:   one of

   ALIAS[ES] { * | alias-name }
   KEY[S] { * | key-name }
   SOURCE ASSIGN { * | original-name }
   SOURCE OPEN[S] { * | source-file }


key-name:   one of
   F1    F2    F3    F4    F5    F6    F7    F8
   F9    F10   F11   F12   F13   F14   F15   F16
   SF1   SF2   SF3   SF4   SF5   SF6   SF7   SF8
   SF9   SF10  SF11  SF12  SF13  SF14  SF15  SF16


original-name:   one of
   [ \system. ] $volume [ .subvolume [ .file ] ]
   [ \system. ] $process [ .#qual-1 [ .qual-2 ] ]
   [ \system. ] cpu, pin
   [ \system. ] $volume.#number
   /oss-pathname [/oss-pathname...]
```

Related Commands

- ADD on page 6-6
- LIST on page 6-129

DELETE ALIAS

The DELETE ALIAS command removes one or all aliases from the alias list for the current Inspect session.

```
DELETE ALIAS[ES] { * | alias-name }
```

*

directs Inspect to remove all aliases.

alias-name

directs Inspect to remove a specific alias.

Usage Consideration

Aliases are not expanded following the keywords DELETE ALIAS.

Related Commands

- ADD ALIAS on page 6-7
- ALIAS on page 6-17
- LIST ALIAS on page 6-131

DELETE KEY

The DELETE KEY command removes one or all function-key definitions from the function-key list for the current Inspect session.

```
DELETE KEY[S] { * | key-name }


key-name:   one of

   F1    F2    F3    F4    F5    F6    F7    F8
   F9    F10   F11   F12   F13   F14   F15   F16
   SF1   SF2   SF3   SF4   SF5   SF6   SF7   SF8
   SF9   SF10  SF11  SF12  SF13  SF14  SF15  SF16
```

*

directs Inspect to remove all function-key definitions.

key-name

directs Inspect to remove a specific function-key definition. Valid key names include F1 through F16 and shifted F1 (SF1) through shifted F16 (SF16).

Related Commands

- ADD KEY on page 6-9
- KEY on page 6-128
- LIST KEY on page 6-136

DELETE SOURCE ASSIGN

The DELETE SOURCE ASSIGN command removes one or all source assignments from the current program's source assignment list.

```
DELETE SOURCE ASSIGN { * | original-name }


original-name:   one of

   [ \system. ] $volume [ .subvolume [ .file ] ]
   [ \system. ] $process [ .#qual-1 [ .qual-2 ] ]
   [ \system. ] cpu, pin
   [ \system. ] $volume.#number
    /oss-pathname [/oss-pathname...]
```

*

directs Inspect to remove all source assignments from the current program's source assignment list.

original-name

directs Inspect to remove a specific source assignment from the current program's source assignment list.

Usage Considerations

- Inspect does not perform partial matching when comparing the original name in DELETE SOURCE ASSIGN with the original names in the source assignments. The original name you specify in DELETE SOURCE ASSIGN must match a source assignment's original name exactly before Inspect will remove that assignment.
- OSS pathnames must be absolute pathnames. For example:

```
-PROG-DELETE SOURCE ASSIGN /usr/fred/src/file.c
```

Related Commands

- ADD SOURCE ASSIGN on page 6-14
- LIST SOURCE ASSIGN on page 6-141
- SOURCE ASSIGN on page 6-202

DELETE SOURCE OPEN

The DELETE SOURCE OPEN command closes one or all source files that Inspect had opened as the result of previous SOURCE commands.

```
DELETE SOURCE OPEN[S] { * | source-file }
```

`*`

directs Inspect to close all open source files.

`source-file`

directs Inspect to close a specific source file.

Usage Considerations

- Inspect opens source files in protected mode to prohibit modification of a file during an Inspect session. DELETE SOURCE OPEN enables you to close the file so that you can modify it.
- OSS pathnames must be qualified to the root level.
- If Inspect has not opened the given source file, it reports this message:

```
** Inspect error 120 ** Source file is not recognized:  file-name
```

Related Commands

- LIST SOURCE OPEN on page 6-142
- SOURCE on page 6-196
- SOURCE OPEN on page 6-208

DISPLAY

The DISPLAY command formats and displays these types of items:

- Data location, including SPI buffers and tokens
- Program registers (TNS and TNS/R)
- Program code
- Quoted strings
- Expressions (after evaluating them)
- Integral values (as a specified data type)

In addition, the DISPLAY command provides several clauses that enable you to control the size, type, and formatting of the items you display.

The complete syntax for the DISPLAY command and its clauses:

```
DISPLAY item [ , item ]... [ formatting-clause ]


item:   one of

   display-data [ WHOLE ] [ PLAIN ] [ FOR for-spec ]
   display-code [ FOR group-spec ]
   REGISTER register-item [ TYPE display-type ]
   spi-buffer
   spi-token [ AS data-type ] [ FOR for-count ]
   string
   ( expression )
   VALUE value-list [ TYPE display-type ]


display-data:   one of

   identifier
   data-location AS data-type
   data-location TYPE display-type


data-location:   one of

   ( expression ) [ SG ]
   identifier
```

```
display-type:  one of

  CHAR             CRTPID           DEVICE       ENV
  FILENAME         FILENAME32       FIXED        FLOAT
  INT              INT16            INT32        LOCATION
  PROCESS HANDLE   REAL             REAL32       REAL64
  SSID             STRING           SYSTEM       TIMESTAMP
  TIMESTAMP48      TOSVERSION       TRANSID      USERID
  USERNAME


for-spec:

   for-count [ BYTE[S] | WORD[S] | DOUBLE[S] | QUAD[S] ]


display-code:  one of

   scope-path
   [ scope-path ] code-reference

code-reference:  one of

   scope-unit [ FROM source-file ]
   label [ FROM source-file  ]
   #line-number [ (source-file) ]
for-count:  one of

   non-negative integer
   data-location


register-item:  one of

 ALL      BOTH      TNS      TNS/R      register-name


register-name:  one of

   tns-register-name
   tns/r-register-name


tns-register-name:  one of

   P      E      L      S
   R0     R1     R2     R3     R4     R5     R6     R7
   RA     RB     RC     RD     RE     RF     RG     RH
```

tns/r-register-name: one of

```
    $0    $1   $2   $3...$31
    $HI   $LO
    $PC
    tns/r-register-alias
```

tns/r-register-alias: one of

```
   $AT   $V0   $V1   $A0   $A1   $A2   $A3
   $S0   $S1   $S2   $S3   $S4   $S5   $S6   $S7
   $T0   $T1   $T2   $T3   $T4   $T5   $T6   $T7   $T8   $T9
   $K0   $K1   $GP   $SP   $FP   $RA
```

spi-buffer:

```
   data-location TYPE spi-type
```

spi-type: one of

```
   EMS      EMS-NUM      SPI      SPI-NUM
```

spi-token:

```
   data-location : token-spec [ TYPE spi-type ]
      [ POSITION token-spec [ , token-spec ]... ]
```

token-spec:

```
   token-code [ : token-index ] [ SSID ssid-string ]
```

token-code: one of

```
   token-index
   ssid-string
```

value-list: one of

```
   integer
   integer , integer
   integer , integer , integer , integer
```

formatting-clause: one of

```
   IN base [ base ]...
   { FORMAT | FMT } format-list
   PIC mask-string [ , mask-string ]...
```

`base`: one of

```
   BINARY    OCT[AL]     DEC[IMAL]   HEX[ADECIMAL]
   ASCII     XASCII      GRAPHIC[S]
   ICODE
```

`format-list`:

an edit-descriptor list for the operating system formatter

`mask-string`:

a mask string for the M edit descriptor

These subsections show you how to use the DISPLAY command to display different types of items:

- Displaying Program Data on page 6-37
- Displaying Program Registers on page 6-42
- Displaying Program Code on page 6-45
- Displaying SPI Data on page 6-48
- Displaying Strings, Expressions, and Constant Values on page 6-58

These subsections are six additional subsections that describe the major clauses of the DISPLAY command:

- Using the AS Clause on page 6-61 shows you how to use the AS clause to display an item using the attributes of a program-defined data type.
- Using the FOR Clause on page 6-64 shows you how to use the FOR clause to display an item for a number of simple or complex groups (for more than two elements).
- Using the FORMAT Clause on page 6-67 shows you how to use the FORMAT clause to format items using the operating system formatter.
- Using the IN Clause on page 6-70 shows you how to use the IN clause to display items in one or more bases.
- Using the PIC Clause on page 6-73 shows you how to use the PIC clause to format items using mask strings and the M edit descriptor of the operating systemþ90 formatter.
- Using the TYPE Clause on page 6-75 shows you how to use the TYPE clause to display an item using the attributes of a predefined data type.

Displaying Program Data

DISPLAY command is used to format and display data in the current program, including SPI buffers and tokens.

Note. Data location is different for different languages. For more information, see language-specific sections.

This subsection shows how to display non-SPI data. Displaying SPI Data on page 6-48 shows how to display SPI data.

```
DISPLAY item [ , item ]... [ formatting-clause ]


item:

   display-data [ WHOLE ] [ PLAIN ] [ FOR for-spec ]


display-data:  one of

   identifier
   data-location AS data-type
   data-location TYPE display-type


data-location:   one of

   ( expression ) [ SG ]
   identifier


formatting-clause:   one of

   IN base [ base ]...
   { FORMAT | FMT } format-list
   PIC mask-string [ , mask-string ]...
```

`item [ , item ]...`

specifies the list of items to display. Inspect determines the default display format of an item based on the type of value that the item represents.

`program-data [ WHOLE ] [ PLAIN ] [ FOR for-spec ]`

directs Inspect to display a value found in the data space of the current program. The type of value that *program-data* represents determines the default unit used by the FOR clause.

The WHOLE clause directs Inspect to display the value as a character string. If the value is a group item (such as a structure or record), Inspect displays it as a single string.

Inspect ignores the WHOLE clause if you use the FOR or IN ICODE clause. The WHOLE clause is also invalid for PATHWAY requester programs.

The PLAIN clause directs Inspect to suppress the identifying information it normally displays for an item, including:

- The name of the item
- The names of the elements of a group item
- The quotation marks that delimit string values

The FOR clause directs Inspect to display the specified number of elements.

For more information, see Using the FOR Clause on page 6-64.

As shown in the diagram, the *`program-data`* parameter is one of these:

```
identifier
data-location AS data-type
data-location TYPE display-type
```

`identifier`

specifies constants and variables defined by the program.

`data-location` `AS` *`data-type`*

specifies the starting address and data type of a data value. The AS clause directs Inspect to display a data value using the attributes of a data type defined in the current program. The parameter *`data-type`* is a data location identifying a data-type definition or a variable. If *`data-type`* refers to a variable, Inspect uses the type attributes of that variable.

The AS clause is invalid for PATHWAY requester programs.

For more information, see Using the AS Clause on page 6-61.

`data-location` `TYPE` *`display-type`*

specifies the starting address and display type of a data value. The TYPE clause causes Inspect to display a data value using the attributes of a specific display type.

`data-location`

specifies the memory location starting at which data is to be formatted as data-type.

`(` *`expression`* `)`

specifies a memory address. If the expression evaluates to a 16-bit value, this value is interpreted as a 16-bit word address. If the expression evaluates to a 32-bit value, this value is interpreted as an extended address.

`SG`

specifies that the 16-bit expression is interpreted as a System Global address. SG is not allowed with 32-bit addresses.

`identifier`

specifies a variable defined by the program. The form of identifier allowed is any legal variable specification, such as i, struct.r1.r2, or a[30].

For more information, see Using the TYPE Clause on page 6-75.

`formatting-clause`

specifies the format Inspect should use when displaying each item. Inspect provides three format clauses: FORMAT, IN, and PIC. For more information about these clauses, see Using the FORMAT Clause on page 6-67, Using the IN Clause on page 6-70, and Using the PIC Clause on page 6-73.

Default Value

If you do not specify formatting-clause, Inspect displays numeric values in the current output radix and delimits character and string values with quotes (").

Usage Considerations

- Displaying Items in a Running Program

  If you enter a DISPLAY command while the current program is running, the data values that Inspect displays might not be valid.

- Displaying FILLER Data Items

  Inspect does not display FILLER elements in group data items unless you use the WHOLE clause, the FOR clause, or one of the formatting clauses.

- Displaying Records

  If the item being displayed is a record or structure, the item name and its components are displayed. Components follow the record name and are indented to denote inclusion. For example, assume this COBOL definition:

```
01 person-name.
   03 last-n PICTURE X(20).
   03 rest   PICTURE X(20).
```

Inspect then displays this item as follows:

```
-PRG-DISPLAY person-name
PERSON-NAME =
  LAST-N = "DINGERDORFF         "
  REST = "MOSTOK R.           "
```

- Displaying Arrays

Items declared as arrays are displayed using the subscript range specified in the command. If none is specified, then the items are displayed in their entirety.

A one-dimensional item is displayed as a row of values, the name indicating the beginning subscript value. For example, assume this COBOL definition:

```
01 person-name.
   03 last-n.
      05 last-char PICTURE X OCCURS 20 TIMES.
   03 rest.
      05 first-char PICTURE X OCCURS 20 TIMES.
```

Inspect then displays this item as follows:

```
-PRG-DISPLAY person-name
PERSON-NAME =
  LAST-N =
      LAST-CHAR[1] = "DINGERDORFF         "
  REST =
     FIRST-CHAR[1] = "MOSTOK R.           "
```

Examples

These examples display program data, based on this source code.

```
?inspect,symbols


STRUCT S^Def ( * );
BEGIN

   INT     one^field;
   INT     two^field;
END;

   PROC example MAIN;
BEGIN
   INT     arr[0:7] := [ 12345, 5, 123, 2, 901, 89, 567, 0 ];
   STRING  s1 [0:6];   STRUCT  def^one(S^Def);
   STRING  .string_ptr;

   def^one.one^field := 1;
   def^one.two^field := 22;

   s1 ':=' "example";
   @string_ptr := @s1;

END;
```

1. This example shows the result of displaying an array of 8 elements.

```
-EX2OBJ-DISPLAY arr
ARR[0] = 12345 5 123 2 901 89 567 0
```

2. This example shows the usage of the WHOLE clause with the DISPLAY command on the same array of 8 elements. Specifying the WHOLE clause will cause the elements in the array to be displayed in string format.

```
-EX2OBJ-DISPLAY arr WHOLE
ARR[0] = "09" ?0 ?5 ?0 "{" ?0 ?2 ?3 ?133 ?0 "Y" ?2 "7" ?0 ?0
```

3. This example shows the usage of the PLAIN clause with the DISPLAY command on the same array of 8 elements. Notice that by specifying the PLAIN clause, the name of the item, in this case arr, was not displayed.

```
-EX2OBJ-DISPLAY arr PLAIN
12345 5 123 2 901 89 567 0
```

4. This example shows the usage of the FOR clause with the DISPLAY command.

```
-EX2OBJ-DISPLAY arr FOR 5
ARR[0] = 12345 5 123 2 901
```

5. This example shows the usage of the DISPLAY command on a structure with two fields.

```
-PROGRAM-DISPLAY def^one
DEF^ONE =
  ONE^FIELD = 1
  TWO^FIELD = 22

-PROGRAM-DISPLAY def^one.two^field
DEF^ONE.TWO^FIELD = 22
```

6. This example shows the usage of the AS clause with the DISPLAY command.

```
-PROGRAM-DISPLAY arr AS s^def
S^DEF =
  ONE^FIELD = 12345
  TWO^FIELD = 5
```

7. This example shows the display of a string array.

```
T-PROGRAM-DISPLAY s1
S1[0] = "example"
```

8. This example shows the display of a string pointer.

```
-PROGRAM-DISPLAY string_ptr
STRING_PTR = "e"
-PROGRAM-DISPLAY string_ptr[0:6]
STRING_PTR[0] = "example"
-PROGRAM-DISPLAY string_ptr FOR 7 BYTES
STRING_PTR[0] = "example"
```

Displaying Program Registers

You can use the DISPLAY command to format and display registers in the current program. For accelerated programs, you can use this command to display the values of TNS machine registers.

Note. The DISPLAY REGISTER command is invalid for PATHWAY requester programs.

```
DISPLAY item [ , item ]... [ formatting-clause ]


item:

   REGISTER register-item [ TYPE display-type ]


register-item: one of
   ALL
   BOTH
   TNS
   TNS/R
   register-name


register-name:   one of

   tns-register-name
   tns/r-register-name


 tns-register-name:  one of

   P      E      L      S
   R0     R1     R2     R3     R4     R5     R6     R7
   RA     RB     RC     RD     RE     RF     RG     RH


tns/r-register-name:  one of

   $0   $1   $2   $3...$31
   $HI   $LO
   $PC
   tns/r-register-alias
```

```
tns/r-register-alias:   one of

   $AT   $V0   $V1   $A0   $A1   $A2   $A3
   $S0   $S1   $S2   $S3   $S4   $S5   $S6   $S7
   $T0   $T1   $T2   $T3   $T4   $T5   $T6   $T7   $T8   $T9
   $K0   $K1   $GP   $SP   $FP   $RA


  formatting-clause:   one of
   IN base [ base ]...
   { FORMAT | FMT } format-list
   PIC mask-string [ , mask-string ]...
```

item [, *item*]...

specifies the list of items to display. Inspect determines the default display format of an item based on the type of value that the item represents.

REGISTER *register-item* [TYPE *display-type*]

directs Inspect to display the current value of a program register.

The TYPE clause causes Inspect to display the register using the attributes of a specific display type.

For more information, see Using the TYPE Clause on page 6-75.

register-item:

is the name of a TNS or TNS/R register.

ALL

displays the default set of machine registers.

BOTH

when debugging a program on a TNS/R system displays both TNS and TNS/R machine registers. To display TNS/E registers when debugging any program on a TNS/E system, you must use Visual Inspect.

TNS

is specified to display TNS machine registers.

TNS/R

is specified when debugging a program on a TNS/R system to display the values of the TNS/R machine registers.

formatting-clause

specifies the format Inspect should use when displaying each item. Inspect provides three format clauses: FORMAT, IN, and PIC. For more information about these clauses, see Using the FORMAT Clause on page 6-67, Using the IN Clause on page 6-70, and Using the PIC Clause on page 6-73.

Usage Considerations

- When debugging an emulated TNS program on a TNS/E system, the DISPLAY command cannot display TNS/E registers. You must use either Visual Inspect or Native Inspect to display TNS/E registers.
- When you use the TYPE clause with the DISPLAY REGISTER command:
    - For TNS registers, display-type values must be 2-byte.
    - For TNS/R registers, display-type values must be 2-byte or 4-byte.
- When listing all TNS registers, the register that is at the top of the register stack is marked with “<--”.
- If an accelerated program is not executing TNS/R instructions or the TNS/R or BOTH clause is used when debugging a program that has not been processed by the Axcel accelerator, this error message is displayed:

```
** Inspect error 371 **  Program is not executing TNS/R instructions
```

- The virtual frame pointer (VFP) for native stack frames is displayed. In addition, the contents of the registers will be relative to the current stack frame. In other words, as the current scope is changed to an active stack frame, the register values will be what they were in that stack frame. If the current scope is changed to a non-active stack frame, the register values will be what are current in the program. Note that not all registers are saved in every procedure. This may result in some registers having values left over from the preceding stack frame.

Output

This example output illustrates all registers.

```
-PROGRAM-DISPLAY REGISTER ALL

     $PC:  1879048880  $HI:      926     $LO:  3976303904  VFP:  1342177056

 $0                  0 $AT:   134217732  $V0:           0  $V1:           1
 $4: $A0             0 $A1            0  $A2    134230564  $A3:          92
 $8: $T0     134230560 $T1            4  $T2           66  $T3   1879082720
$12: $T4         65043 $T5       262144  $T6            0  $T7            0
$16: $S0    4294967295 $S1   4294967295  $S2   4294967295  $S3   4294967295
$20: $S4    4294967295 $S5   4294967295  $S6   4294967295  $S7   4294967295
$24: $T8     134230656 $T9            2  $K0   2803083027  $K1   2803083027
$28: $GP     134254384 $SP   1342177016  $FP   4294967295  $RA   1879082720
```

Examples

1. This command displays all TNS registers in octal:

```
-C000OTCE-DISPLAY REGISTER ALL IN OCTAL
R0    %077777  RH     P = %003105
R1    %174030  RG     E = %000207  (RP=7,CCG,T)
R2    %105260  RF     L = %005022
R3    %144402  RE
R4    %107571  RD     S = %005034
R5    %011674  RC
R6    %160413  RB     SPACE = %000000  UC.0
R7    %062400  RA <--
```

2. This command displays TNS/R registers in hexadecimal:

```
-C000OTCE-DISPLAY REGISTER TNS/R IN HEXADECIMAL
      $PC: %H70421D60  $HI: %H7FFFFC0B  $LO: %H74FB5D47  VFP: %H729F2176
 $0:       %H00000000  $AT: %H70000000  $V0: %H00000000  $V1: %H7FFFF818
 $4:  $A0: %H7FFFF818  $A1: %H00000003  $A2: %H00000645  $A3: %H74FB5D47
 $8:  $T0: %H70421D60  $T1: %H70421D44  $T2: %H70400000  $T3: %H70400000
$12:  $T4: %H00000002  $T5: %H00000002  $T6: %H00000000  $T7: %H7FFFF819
$16:  $S0: %H00007FFF  $S1: %HFFFFF818  $S2: %HFFFF8AB0  $S3: %HFFFFC902
$20:  $S4: %H0A548F79  $S5: %H000013BC  $S6: %HDD7EE10B  $S7: %H00006500
$24:  $T8: %H70000000  $T9: %H00000080  $K0: %HA713A713  $K1: %HA713A713
$28:  $GP: %H70401000  $SP: %H00001438  $FP: %H00001424  $RA: %H724201D4
```

Displaying Program Code

You can use the DISPLAY command to format and display object code in the current program.

Note. You cannot display the pseudocode in a PATHWAY requester program.

```
DISPLAY item [ , item ]... [ formatting-clause ]


item:

   display-code  [ FOR for-spec ]


display-code:   one of

    scope-path
   [ scope-path ] code-reference


code-reference:   one of

   scope-unit [ FROM source-file ]
   label [ FROM source-file  ]
   #line-number [ (source-file)
```

```
formatting-clause:   one of

   IN base [ base ]...
   { FORMAT | FMT } format-list
   PIC mask-string [ , mask-string ]...
```

item [, item]

specifies the list of items to display. Inspect determines the default display format of an item based on the type of value that the item represents.

display-code

specifies a code location to display. *scope-path* and *code-reference*are described in Section 2, Inspect Concepts.

WHOLE

directs Inspect to display the value as a character string. Inspect ignores the WHOLE clause if you use the FOR or IN ICODE clause.

PLAIN

directs Inspect to suppress the identifying information it normally displays for an item, including:

- The name of the item
- The names of the elements of a group item
- The quotation marks that delimit string values

FOR *for-spec*

directs Inspect to display the value as a number of groups. The FOR clause uses WORD as the default unit when displaying program code. For more information, see Using the FOR Clause on page 6-64.

formatting-clause

specifies the format Inspect should use when displaying each item. Inspect provides three format clauses: FORMAT, IN, and PIC. For more information about these clauses, see Using the FORMAT Clause on page 6-67, Using the IN Clause on page 6-70, and Using the PIC Clause on page 6-73.

Usage Considerations

- You can display code from user library routines if you have read access to the code file.
- The ICODE command provides a more powerful mechanism for displaying program code.

Example

This example illustrates the differences between using the DISPLAY and ICODE commands to display program code. Note that the ICODE output indicates which source line numbers correspond to which instructions. In addition, the ICODE command recognizes where the procedure ends, as opposed to DISPLAY which displays only the exact amount you specified.

```
-PROGRAM-DISPLAY #m IN I FOR 40 WORDS
M =   ADDS   +012       LDI    +065       STOR  L+001       LDI    +146
  STOR  L+002       LADR  L+003       LLS     01        RDP
  LDI    +011       ORLI    000       LADD              LLS     01
  LDI    +013       MOVB    047       ZERD              PUSH    711
  XCAL    000       STOR  G+145       STD   G+154,6     STD   L+040,7
  LDB   L+157,7     LADR  G+154,5     LDD   G+000,6     NOP
  NOP               NOP               NOP               NOP
  NOP               NOP               NOP               NOP
  NOP               NOP               NOP               NOP
  NOP               NOP               NOP               NOP
-PROGRAM-ICODE AT #m FOR 40 STATEMENTS
#9
  %000003:    ADDS   +012
#18
  %000004:    LDI    +065            STOR  L+001
#19
  %000006:    LDI    +146            STOR  L+002
#20
  %000010:    LADR  L+003            LLS     01             RDP
  %000013:    LDI    +011            ORLI    000            LADD
  %000016:    LLS     01             LDI    +013            MOVB    047
#23
  %000021:    ZERD                   PUSH    711            XCAL    000
  %000024:    STOR  G+145            STD   G+154,6          STD   L+040,7
  %000027:    LDB   L+157,7          LADR  G+154,5          LDD   G+000,6
 ** Inspect warning 388 **  Listing ends at procedure end
```

Displaying SPI Data

You can use the DISPLAY command to format and display SPI buffers and SPI tokens in the current program.

```
DISPLAY item [ , item ]... [ IN base [ base ]... ]


item:   one of

   spi-buffer
   spi-token [ AS data-type ] [ FOR for-count ]


spi-buffer:

   data-location TYPE spi-type


data-location:   one of

   ( expression ) [ SG ]
   identifier


spi-type:   one of

   EMS   EMS-NUM   SPI   SPI-NUM


spi-token:

   data-location : token-spec [ TYPE spi-type ]
      [ POSITION token-spec [ , token-spec ]... ]


token-spec:

   token-code [ : token-index ] [ SSID ssid-string ]
```

item [, *item*]...

specifies the list of items to display. Inspect determines the default display format of an item based on the type of value that the item represents.

spi-buffer

directs Inspect to display an entire SPI buffer. The syntax of *spi-buffer* is:

data-location TYPE *spi-type*

The *`data-location`* parameter specifies the location of the SPI buffer, and the TYPE clause specifies how you want Inspect to interpret and format the buffer. Here are the SPI types that Inspect supports:

- EMS directs Inspect to interpret the buffer as an EMS buffer and presents its contents in a labelled format.
- EMS-NUM directs Inspect to interpret the buffer as an EMS buffer and presents its contents in a numeric format.
- SPI directs Inspect to interpret the buffer as an SPI buffer and presents its contents in a labelled format.
- SPI-NUM directs Inspect to interpret the buffer as an SPI buffer and presents its contents in a numeric format.

`data-location`

specifies the memory location starting at which data is to be formatted as *`data-type`*.

`(` *`expression`* `)`

specifies a memory address. If the expression evaluates to a 16-bit value, this value is interpreted as a 16-bit word address. If the expression evaluates to a 32-bit value, this value is interpreted as an extended address.

`SG`

specifies that the 16-bit expression is interpreted as a System Global address. SG is not allowed with 32-bit addresses.

`identifier`

specifies a variable defined by the program. The form of *`identifier`* allowed is any legal variable specification, such as i*, struct.r1.r2*, or *a[30]*.

uses the indirect address as a system global byte address.

`spi-token` `[ AS` *`data-type`* `] [ FOR` *`for-count`* `]`

directs Inspect to display an individual token or list within an SPI buffer.

The AS clause causes Inspect to display a data value using the attributes of a data type defined in the current program. The parameter *data-type* is a data location identifying a data-type definition or a variable. If *data-type* refers to a variable, Inspect uses the type attributes of that variable.

The FOR clause directs Inspect to display a given number of occurrences of the token or list within its containing SPI buffer.

spi-token

specifies the location of the SPI token or list. The syntax for *spi-token* is:

```
data-location : token-spec [ TYPE spi-type ]
   [ POSITION token-spec [ , token-spec ]... ]
```

The data-location parameter specifies the location of the SPI buffer containing the token or list, and the TYPE clause specifies how you want Inspect to interpret and format the buffer.

For the SPI types that Inspect supports refer to the *spi-type* discussion above.

The *token-spec* parameter identifies the token or list for Inspect to display. If you specify a list token, Inspect displays all tokens in the list.

The POSITION clause specifies the position of the token in the buffer. When used with the POSITION clause, *token-spec* provides a path to a token that is within one or more lists.

token-spec

identifies the token or list for Inspect to display. If you specify a list token, Inspect displays all tokens in the list. The syntax for *token-spec* is:

```
token-code [ : token-index ] [ SSID ssid-string ]
```

The token-code parameter is an integer that specifies the token type and token number of the token to display; it can be a 32bit value or a data location containing a 32-bit value. Inspect passes this value to the SSGET or EMSGET procedure.

For more information about command and response message buffers, see the *DSM/SCM Event Management Programming Manual.*

The *token-index* parameter is an integer that specifies a specific occurrence of the token indicated by *token-code*; it can be a 16-bit value or a data location containing a 16bit value. Inspect passes *token-index* to an SSGET or EMSGET procedure.

Here are the possible values for token-index:

- Inspect displays the next occurrence of the token after the current position in the buffer. The current position marks the last token selected from the buffer by the application with an SSGET procedure. For more information about the current position, see the *DSM/SCM Event Management Programming Manual.*
- Inspect displays the nth occurrence of the token.

If you do not specify a *token-index*, Inspect displays all occurrences of the token.

`SSID` *`ssid-string`*

specifies a subsystem ID for the token or list. The *`said-string`* parameter is a string enclosed by quotation marks. It has one of these forms:

```
"owner.subsys-name.version"
"owner.subsys-number.version"
```

`owner` is an eight-character ASCII string that identifies the name of the company or organization providing the definition for the token. Owner corresponds to the Z-OWNER field in the subsystem ID structure.

For NonStop subsystems, *`owner`* is “TANDEM.” Users select a name of their own when defining their tokens.

`subsys-name` specifies the subsystem name for the token. Examples are FUP, PUP, and TMF.

The *`subsys-number`* parameter is a signed integer value that identifies the subsystem. *`subsys-number`* corresponds to the Z-NUMBER field in the subsystem ID structure.

The subsystem owner provides a subsystem number for each subsystem. For Tandem subsystems, the subsystem numbers are in the ZSPIDEF.ZSPIDDL file.

`version` is the software release version of the subsystem. Examples are C00 and C10. *`version`* corresponds to the Z-VERSION field in the subsystem ID structure.

Note that *`owner, subsystem-name,`* and *`version`* are case-sensitive and must be entered as they are defined.

`IN` *`base`* `[` *`base`* `] ...`

directs Inspect to display each item in one or more bases. For more information, see Using the IN Clause on page 6-70.

General Usage Considerations

- Displaying Items in a Running Program

  If you enter a DISPLAY command while the current program is running, the data values that Inspect displays might not be valid.

- Displaying SPI Data

  To ensure that displaying SPI data doesn't affect the position pointers for an SPI buffer, Inspect does not extract information from the actual SPI buffer. Instead, it makes a private copy of the SPI buffer and extracts information from this copy.

- SPI Types in the TYPE Clause

  If you specify SPI or SPI-NUM as the SPI type, Inspect uses the SSGET system procedure to extract information from the SPI command or response buffer. If you specify EMS or EMS-NUM as the SPI type, Inspect uses the EMSGET system procedure to extract information from the event-message buffer. If the buffer is a command or response buffer, Inspect uses SSGET, even if you specify EMS or EMS-NUM as the SPI type.

  For more information about command and response message buffers, see the *DSM/SCM Event Management Programming Manual.* For more information about event-message buffers, see the *Event Management Service (EMS) Analyzer Manual* (formerly the *Event Management System (EMS) Manual*).

Usage Considerations When Displaying an SPI Buffer

- Inspect displays each token in the buffer sequentially; header tokens appear first and the remaining tokens next. The maximum buffer length is shown after the last header token as:

```
BUFFER LENGTH = max-bytes
```

  where `max-bytes` is the maximum number of bytes in the buffer. This value is taken from the Z-BUFLEN field in the buffer.

- When you specify EMSNUM or SPINUM as the SPI type, Inspect can display a buffer up to 4K bytes in length. When you specify EMS or SPI as the SPI type, Inspect can display a buffer up to 64K bytes in length. If you attempt to display a buffer that is too large, Inspect displays this error message:

```
** Inspect error 189 **  SPI buffer too large
```

- Inspect displays the subsystem ID for each token and the data-length word for each variable-length token. Inspect also marks two specific tokens in the buffer with these special characters:

*	Marks the token at the current position in the buffer. This token was the last token that your program selected by calling an SSGET or EMSGET procedure.
-	Marks the token at the last position in the buffer. This token was the last token added to the buffer with an SSPUT procedure.

- Data lists, error lists, and generic lists are marked at the beginning and end of each list, as shown in Table 6-2 on page 6-53. Tokens within a list are indented two spaces.

Usage Considerations When Displaying an SPI Token or List

- Inspect always points to the start of the SPI buffer, unless you specify zero for token-index or the POSITION clause. If you access the SPI buffer with Inspect, the current position in the SPI buffer does not change. If you specify zero for token-index, Inspect searches for the requested token beginning at the current position in the buffer.

  The current position points to the last token selected by an SSGET procedure in a program and is marked by an asterisk when you display the entire buffer. For more information about the current position, see the *DSM/SCM Event Management Programming Manual*.

- You must specify the subsystem ID for a token whenever the subsystem ID for the token differs from the current subsystem ID. The current subsystem ID is the subsystem ID of the SPI buffer, unless you use the POSITION clause to select a token within a list; in this case, the current subsystem ID is that of the list. For more information about the subsystem ID, see the *DSM/SCM Event Management Programming Manual.*

- If you use the AS clause with a variable-length token, you must ensure that the template specified for `data-type` matches the actual data.

- You can use the symbolic names of tokens as defined by any subsystem (including SPI or EMS tokens) to display data from an SPI buffer, provided that the names have been defined as symbols to the compiler. To use tokens from a specific subsystem, you must compile the source file containing the tokens with the ?SYMBOLS compiler directive. For example, a COBOL85 application that uses FUP definitions must include the ZFUPCOB file during compilation.

- To display a token within a list that is within a second list, use the POSITION clause with two `token-spec` parameters. The first `token-spec` parameter positions within the first list, and the second `token-spec` parameter positions within the second list.

Table 6-2. SPI Token Formatting by the DISPLAY Command (page 1 of 2)

Token Type	Formatting Description
BOOLEAN	F if zero (0);T if a value other than zero
BYTE, UNIT	Unsigned number
CHAR	Enclosed in quotation marks if displayable; otherwise, as an unsigned numeric byte value preceded by a question mark
CRTPID	[\system.][$*process-name*]*cpu,pin*
DEVICE	[\system.]$*device*
ENUM	Signed integer
ERROR	Subsystem ID (as described under SSID), followed by a signed integer for the error number
FLT, FLT2	Signed floating-point

Table 6-2. SPI Token Formatting by the DISPLAY Command (page 2 of 2)

Token Type	Formatting Description
FNAME	[\system.]
FNAME32	\system.$*volume.subvolume.filename*
INT, INT2, INT4	Signed integer
LIST	The word LIST for a generic list, the words ERROR LIST for an error list, or the words DATA LIST for a data list
MAP	Enclosed in quotation marks if displayable; otherwise, as an unsigned numeric byte value preceded by a question mark
MARK	The word MARK or the words SUBJECT MARK for the ZEMS-SUBJECT-MARK token
SSCTL	The words END LIST
SSID	*owner.ssname.version-text* if the SSNAME record for the subsystem ID is available, or *owner.ssnumber.version-number* converted using the SSIDTOTEXT procedure
SSTBL	The word TABLE
STRUCT	Enclosed in quotation marks if displayable; otherwise, as an unsigned numeric byte value preceded by a question mark
SUBVOL	*[system.]$volume.subvolume.*
TIMESTAMP	*YYYY-MM-DD HH:MM:SS.mmm.nnn* timestamp converted from GMT to local civil time (LCT), or, if the timestamp is less than year one before the conversion to LCT, as the elapsed time in microseconds
TOKENCODE	32-bit signed number for the token code, followed by the token data type, token length, and token number in parentheses
TRANSID	TMF transaction ID converted using the TRANSIDTOTEXT procedure
USERNAME	*group.user group-id, user-id*

Examples that Display SPI Buffers

The symbolic names in the following examples are in DDL format, which uses hyphens as separators. If your application is written in TAL, Inspect displays symbolic names using the circumflex symbol (^) rather than the hyphens.

1. The following command displays an SPI response buffer named SPI-BUFFER. Inspect uses the SSGET procedure to extract the information from the buffer. The

asterisk marks the token at the current position in the buffer. In this example, a parameter was missing from the command buffer:

```
-OBJ-DISPLAY spi-buffer TYPE SPI-NUM
ZSPI-TKN-HDRTYPE = 0 (ZSPI-VAL-CMDDHR)
ZSPI-TKN-CHECKSUM = 0
ZSPI-TKN-COMMAND = 2
ZSPI-TKN-LASTERR = -8 (ZSPI-TKN-MISTKN) - Token not found
ZSPI-TKN-LASTERRCODE = 486866497 (29,4,-506)
ZSPI-TKN-MAX-FIELD-VERSION = 0
ZSPI-TKN-MAX-RESP = 0
ZSPI-TKN-OBJECT-TYPE = 13
ZSPI-TKN-SERVER-VERSION = 17152 C10
ZSPI-TKN-SSID = TANDEM.DNS.C10
ZSPI-TKN-USEDLEN = 128
BUFFER LENGTH = 4196

TANDEM.DNS.C10        (11,2,7) = 26
TANDEM.DNS.C10       *(11,2,0) = -33
TANDEM.DNS.C10        (37,0,-252) - ERROR LIST
TANDEM.DNS.C10          (28,14,-251) = TANDEM.DNS.C10 33
TANDEM.DNS.C10          (7,255,-250) - LENGTH 3 = ?1 ?0 ?1
TANDEM.DNS.C10          (11,2,78) = 2
TANDEM.DNS.C10          (2,4,77) = 5
TANDEM.DNS.C10        (39,0,-254) - END LIST
```

2. This command displays an event-message buffer named EVENT-BUFFER. Inspect uses the EMSGET procedure to extract information from the buffer:

```
-OBJ-DISPLAY event-buffer TYPE EMS-NUM
ZSPI-TKN-HDRTYPE = 1 (ZSPI-VAL-EVTHDR)
ZSPI-TKN-CHECKSUM = 0
ZSPI-TKN-LASTERR = 0 (ZSPI-ERR-OK) - Operation successful
ZSPI-TKN-LASTERRCODE = 0 (0,0,0)
ZSPI-TKN-MAX-FIELD-VERSION = 0
ZSPI-TKN-SSID = TANDEM.MSGSYS.C00
ZSPI-TKN-USEDLEN = 128
ZEMS-TKN-CONSOLE-PRINT = 0 (FALSE)
ZEMS-TKN-CPU = 10
ZEMS-TKN-CRTPID = \COMM.10,0
ZEMS-TKN-EMPHASIS = 0 (FALSE)
ZEMS-TKN-EVENTNUMBER = 104
ZEMS-TKN-GENTIME = 1987-09-04 17:48:13.993.437
ZEMS-TKN-LOGTIME = 1987-09-04 17:48:16.447.002
ZEMS-TKN-PIN = 0ZEMS-TKN-SUPPRESS-DISPLAY = 0 (FALSE)
ZEMS-TKN-SYSTEM = 116 \COMM
ZEMS-TKN-USERID = 255,255 SYSTEM.MANAGERBUFFER LENGTH = 128

TANDEM.MSGSYS.0      *(31,0,-523) - SUBJECT MARK
TANDEM.MSGSYS.0       (9,2,2) = 10
TANDEM.MSGSYS.0       (9,2,4) = 15
TANDEM.GUARDLIB.0     (12,2,26) = 77 0
TANDEM.GUARDLIB.0    -(9,2,25) = 3
```

3. This command displays a user-defined command and response buffer named SPI-BUFFER:

```
-OBJ-DISPLAY spi-buffer TYPE SPI-NUM
ZSPI-TKN-HDRTYPE = 0 (ZSPI-VAL-CMDHDR)
ZSPI-TKN-CHECKSUM = 0
ZSPI-TKN-COMMAND = 2
ZSPI-TKN-LASTERR = 0 (ZSPI-ERR-OK) - Operation successful
ZSPI-TKN-LASTERRCODE = 486866497 (29,4,-506)
ZSPI-TKN-MAX-FIELD-VERSION = 0
ZSPI-TKN-MAX-RESP = 0
ZSPI-TKN-OBJECT-TYPE = 10
ZSPI-TKN-SERVER-VERSION = 0
ZSPI-TKN-SSID = CUST.600.0
ZSPI-TKN-USEDLEN = 250
BUFFER LENGTH = 2048

CUST.600.0       (2,2,6020) = 42
UST.600.0       (20,24,6010) = $DSK02.VOLSPI.COMMUP
CUST.600.0       (7,255,6055) - LENGTH 4 = ?0 ?1 ?128 ?255
CUST.600.0       (7,255,6055) - LENGTH 12 = ?0 ?0 ?0 ?0 ?0 ?0 ?0 ?0 ?0 ?0 ?0
?0
CUST.600.0       (10,2,6005) =  T
CUST.600.0       (10,2,6005) =
CUST.600.0       (28,14,6095) = CUST.600.0 -1
CUST.600,0       (23,8,6060) = 123 milliseconds 456 microseconds
CUST.600.0      *(23,8,6060) = 1988-03-08 21:00:00.000.000
CUST.600.0       (37,0,6253) - LIST
CUST.600.0       (22,8,6090) = \SYS02.$CCEG 11,106
CUST.600.0       (21,16,6085) = $CLIFF
CUST.600.0       (2,6,6075) = 10 5 27
CUST.600.0       (39,0,6254) - END LIST
CUST.600.0      -(1,255,6040) - LENGTH 14 = "Cust Computers"
```

Examples that Display SPI Tokens and Lists

The symbolic names in the following examples are in DDL format, which uses hyphens as separators. If your application is written in TAL, Inspect displays symbolic names using the circumflex symbol (^) rather than the hyphens.

1. This command displays a timestamp token from SPI-BUFFER:

```
-OBJ-DISPLAY spi-buffer:zems-tkn-gentime
TANDEM.PUP.C10      (23,8,-514) = 1988-04-15 10:16:28.092.163
```

2. This command displays all occurrences (the default) of a token from SPIBUFFER:

```
-OBJ-DISPLAY spi-buffer:zfup-tkn-source-file
TANDEM.FUP.C00    (25,32,2)[1] = \SYS2.$DSK2.SAL88.TAB1
TANDEM.FUP.C00    (25,32,2)[2] = \SYS2.$DSK2.SAL88.TAB2
TANDEM.FUP.C00    (25,32,2)[3] = \SYS2.$DSK2.SAL88.TAB3
TANDEM.FUP.C00    (25,32,2)[4] = \SYS2.$DSK2.SAL88.TAB4
```

3. This command displays the default subsystem ID for SPIBUFFER:

```
-OBJ-DISPLAY spi-buffer:zspi-tkn-ssid
TANDEM.TMF.C00        (24,12,-505) = TANDEM.TMF.C00
```

4. This command displays a token within an error list that is also within a second error list in SPI-BUFFER. The first POSITION parameter positions Inspect within the first error list, and the second POSITION parameter positions Inspect within the second error list. The token ZSPI-TKN-ERROR shows file-system errorþ11 (file is not in the directory):

```
-OBJ-DISPLAY spi-buffer:zspi-tkn-error &
POSITION zspi-tkn-errlist, zspi-tkn-errlist
TANDEM.FUP.C10        (28,14,-251) = TANDEM.68.C10 11
```

5. This command displays a data list from SPI-BUFFER. The display includes all tokens within the list; each token is indented two spaces:

```
-OBJ-DISPLAY spi-buffer:zspi-tkn-datalist
TANDEM.FUP.C00      (37,0,-253) - DATA LIST
TANDEM.FUP.C00        (25,32,2) = \SYS1.$DSK1.PAY.ACCTS
TANDEM.FUP.C00        (11,2,0) = 0
TANDEM.FUP.C00       (39,0,-254) - END LIST
```

6. This command displays a user-defined token from SPIBUFFER using a data type named MESSAGE-DEF:

```
-OBJ-DISPLAY spi-buffer:cust-abend-message AS message-def
MESSAGE-DEF =
  MESSAGE-LENGTH = 30
  MESSAGE-TEXT = "$BM01: ERROR - PROCESS ABENDED"
```

7. This command displays a user-defined token within an error list from SPI-BUFFER. The subsystem ID of the token is different from the subsystem ID of the error list, and the subsystem ID of the error list is different from the default subsystem ID for the buffer.

   The POSITION clause directs Inspect to find the token within the error list, and the SSID clause specifies the subsystem ID for the token and error list:

```
-OBJ-DISPLAY spi-buffer:cust2-err-ipc-error SSID "CUST2.600.0" &
POSITION cust1-tkn-errlist SSID "CUST1.500.0"
CUST2.600.0        (28,14,6251) = CUST2.600.0 200
```

Displaying Strings, Expressions, and Constant Values

You can use the DISPLAY command to format and display strings, expressions (after Inspect evaluates them), and constant values. You can use the FORMAT, IN, and PIC clauses to format any of these items. You can also use the TYPE clause to format constant values.

```
DISPLAY item [ , item ]... [ formatting-clause ]


item:  one of

   string
   ( expression )
   VALUE value-list [ TYPE display-type ]


value-list:  one of
   integer
   integer , integer
   integer , integer , integer , integer


formatting-clause:  one of
   IN base [ base ]...
   { FORMAT | FMT } format-list
   PIC mask-string [ , mask-string ]...
```

item [, *item*]...

specifies the list of items to display. Inspect determines the default display format of an item based on the type of value that the item represents.

string

specifies a string of characters to display. This string is a group of zero or more characters enclosed in either quotes or apostrophes. To include a quote in a quote-delimited string, use a pair of quotes. Likewise, to include an apostrophe in an apostrophe-delimited string, use a pair of apostrophes.

(*expression*)

specifies an expression to evaluate and display. The expression must use the syntax appropriate for the current source language. Note that this is not true for “@” used in an expression in COBOL85. For more information, see Section 10, Using Inspect With COBOL and SCREEN COBOL.

`VALUE` *`value-list`* `[ TYPE` *`display-type`* `]`

specifies a list of one, two, or four integer values to display. The TYPE clause causes Inspect to display the values using the attributes of a specific display type. For more information about the TYPE clause, see *`formatting-clause`*

specifies the format Inspect should use when displaying each item. Inspect provides three format clauses: FORMAT, IN, and PIC. For more information, see Using the FORMAT Clause on page 6-67, Using the IN Clause on page 6-70, and Using the PIC Clause on page 6-73.

Usage Guidelines

- Displaying Strings

  If you are logging your Inspect session to a disk file, you might want to display a character string when the program reaches a certain point in its execution. Here is an example of using a DISPLAY command to report only a character string:

```
-PRG-BREAK stage-two THEN "DISPLAY 'Finished stage 1';RESUME"
```

- Using TYPE with VALUE

  When you use the VALUE clause with the TYPE clause, you must ensure that the size of the values you provide matches the size of the display type you specify.

- Displaying Types

  Display types CRTPID, DEVICE, FILENAME, FILENAME32, PROCESS HANDLE, SSID, and USERNAME may not be used with VALUE.

Examples

1. This example shows converting two integer values into an INT(32) value.

```
--DISPLAY VALUE %10,0
524288
```

2. This example illustrates converting four integer values into a quad word value.

```
--DISPLAY VALUE 748,50977,49935,40960
 210762230400000000.
```

3. This example illustrates using the VALUE clause with the TYPE clause for ENV.

```
--DISPLAY VALUE %4207 TYPE ENV
(RP=7,CCG,T,LS)   UL.7
```

4. This shows a legal and illegal system value. If the system number is not legal or if it does not match a system number on the current network, Inspect displays

```
--DISPLAY VALUE 42 TYPE SYSTEM
\SYS
--DISPLAY VALUE 12345 TYPE SYSTEM
\??
```

5. This example illustrates using the VALUE clause with the TYPE clause for TIMESTAMP48.

```
--DISPLAY VALUE 0,8,62563,12864 TYPE TIMESTAMP48
1987-03-09 09:00:00.00
```

6. This example illustrates using the VALUE clause with the TYPE clause for TIMESTAMP64.

```
--DISPLAY VALUE 748,50977,49935,40960 TYPE TIMESTAMP64
1966-09-08 21:00:00.000.000
```

7. This example illustrates using the VALUE clause with the TYPE clause for TOSVERSION.

```
--DISPLAY VALUE 19978 TYPE TOSVERSION
N10
```

8. This example illustrates using the VALUE clause with the TYPE clause for USERID.

```
--DISPLAY VALUE 65535 TYPE USERID
255,255 SUPER.SUPER
```

9. This shows using TYPE LOCATION with and without the space ID. If you omit the space ID, Inspect uses the current space. You must be debugging a program to use TYPE LOCATION.

```
-PROGRAM-DISPLAY VALUE 188 TYPE LOCATION
#CCL.1, #CCL.#700(\BONDS.$DEBUG.CCLSRC.UMCCL), #CCL + %0I
-PROGRAM-DISPLAY VALUE 188 TYPE LOCATION UC.1
#SHOW^TERM.1, #SHOW^TERM.#4919(\BONDS.$DEBUG.CCLSRC.TMCCLSHW), #SHOW^TERM +
%0I
```

10. This example illustrates the displaying of a string constant and an expression.

```
-PROGRAM-DISPLAY ("A") in OCTAL
%101
-PROGRAM-DISPLAY (%777) in BINARY
0000000111111111
```

11. This example illustrates displaying a string.

```
-PROGRAM-DISPLAY "Hello World"
"Hello World"
```

12. This is an example of using an apostrophe in an apostrophe-delimited string.

```
-PROGRAM-DISPLAY 'T''s value is ', T PLAIN
"T's value is ",  53
```

13. This example shows the use of displaying a string with the IN clause.

```
-PROGRAM-DISPLAY "Hello" IN OCTAL
%110 %145 %154 %154 %157
```

14. The next example shows the use of displaying a simple expression.

```
-PROGRAM-DISPLAY (132 + 98)
230
```

15. This example shows the use of expressions with the IN clause.

```
-PROGRAM-DISPLAY (231) IN BINARY
0000000011100111
```

16. This example displays the address of a variable

```
-PROGRAM-DISPLAY (@two_var)
6
```

Using the AS Clause

The AS clause directs Inspect to display an item using the attributes of a data type defined in the current program.

Note. The AS clause is invalid for PATHWAY requester programs.

```
DISPLAY item AS data-type

item:   one of

   data-location
   spi-token

data-location:   one of

   ( expression ) [ SG ]
   identifier
```

item

specifies the item you want to display. You can specify a data address or an SPI token as the item.

`AS` *`data-type`*

directs Inspect to display the item using the attributes of the given data type. The *`data-type`* parameter is a data type identifier or a variable. If *`data-type`* refers to a variable, Inspect uses the type attributes of that variable.

`data-location`

specifies the memory location starting at which data is to be formatted as *`data-type`*.

`(` *`expression`* `)`

specifies a memory address. If the expression evaluates to a 16-bit value, this value is interpreted as a 16-bit word address. If the expression evaluates to a 32-bit value, this value is interpreted as an extended address.

`SG`

specifies that the 16-bit expression is interpreted as a System Global address. SG is not allowed with 32-bit addresses.

`identifier`

specifies a variable defined by the program. The form of identifier allowed is any legal variable specification, such as *`i, struct.r1.r2,`* `or` *`a`*`[`*`30`*`]`.

Examples

This TAL source code was used to generate the examples for the AS clause of the DISPLAY command.

```
?symbols

STRUCT S^Def(*);
  BEGIN
    INT   one_field;
    INT   two_field;
  END;

PROC M MAIN;
BEGIN
  LITERAL STRING_BLEN = 11;
  STRUCT   one_var(S^Def);
  STRING   two_var[0:STRING_BLEN-1];
  INT      three_var[0:7] := [12345, 5, 123, 2, 901, 89, 567, 0];
  STRING  .string_ptr;
  INT     .int_ptr;
  INT(32)  address_ext;
  INT      address_word;
  INT .EXT extended_array[0:1];
```

```
    one_var.one_field := 53;
    one_var.two_field := 102;
    two_var ':=' "Hello World";
    @string_ptr := @two_var;
    @int_ptr := @one_var;
    address_ext := $XADR(one_var);
    address_word := @one_var;
    extended_array ':=' [17, 18];
  END;
```

1. This example uses the AS clause with a structure variable.

```
-PROGRAM-DISPLAY int_ptr AS one_var
ONE_VAR =
  ONE_FIELD = 53
  TWO_FIELD = 102
```

2. This example uses the AS clause with a structure template.

```
-PROGRAM-DISPLAY int_ptr AS s^def
S^DEF =
  ONE_FIELD = 53
  TWO_FIELD = 102
```

3. This example uses the AS clause when a 16-bit word address is stored in an integer variable. An expression that evaluates to a 16-bit value is interpreted as a 16-bit word address.

```
-PROGRAM-DISPLAY (address_word) AS s^def
S^DEF =
  ONE_FIELD = 53
  TWO_FIELD = 102
```

4. This example uses the AS clause when an extended address is stored in an INT(32) variable. An expression that evaluates to a 32-bit value is interpreted as an extended address.

```
-PROGRAM-DISPLAY (address_ext) AS s^def
S^DEF =
  ONE_FIELD = 53
  TWO_FIELD = 102
```

5. This example uses the AS clause when the data address is already known.

```
-PROGRAM-DISPLAY address_word
ADDRESS_WORD = 1
-PROGRAM-DISPLAY address_ext
ADDRESS_EXT = 2
-PROGRAM-DISPLAY (@extended_array)
524294
-PROGRAM-DISPLAY (1) AS s^def
S^DEF =
  ONE_FIELD = 53
  TWO_FIELD = 102
-PROGRAM-DISPLAY (2D) AS s^def
S^DEF =
  ONE_FIELD = 53
  TWO_FIELD = 102
-PROGRAM-DISPLAY (524294) AS s^def
S^DEF =
  ONE_FIELD = 17
  TWO_FIELD = 18
```

6. This example uses the AS clause in conjunction with the IN clause to display the data in ASCII and in an alternate base.

```
-PROGRAM-DISPLAY (address_word) AS s^def IN ASCII
S^DEF =
  ONE_FIELD = ?0 "5"
  TWO_FIELD = ?0 "f"
-PROGRAM-DISPLAY (address_word) AS s^def IN OCTAL
S^DEF =
  ONE_FIELD = %65
  TWO_FIELD = %146
```

7. Since a 16-bit expression is interpreted as a 16-bit word address, you must divide by 2 when the address is a 16-bit byte address.

```
-PROGRAM-DISPLAY (@string_ptr[6]/2) AS s^def IN ASCII
S^DEF =
  ONE_FIELD = "Wo"
  TWO_FIELD = "rl"
```

Using the FOR Clause

The FOR clause directs Inspect to display multiple items of program data or program code. When used to display an SPI token, the FOR clause directs Inspect to display a given number of occurrences of the token within its containing SPI buffer.

Note. The FOR clause is invalid for PATHWAY requester programs.

```
        { display-data FOR for-spec }
DISPLAY { display-code FOR for-spec }
        { spi-token FOR for-count        }


for-spec:

   for-count [ BYTE[S] | WORD[S] | DOUBLE[S] | QUAD[S] ]


for-count:   one of

   expression
   data-location
```

for-spec

defines the format of the groups as a specific number of units. To specify the number, you can use an integer or a data location that contains an integer value. When specifying the units, you must use one of these:

`BYTE[S]   WORD[S]   DOUBLE[S]   QUAD[S]`

If you do not specify a unit, Inspect uses the default unit attributes of the item being displayed.

for-count

specifies how many units or occurrences of an SPI token to display. To specify the count, you can use an expression or a data location that contains an integer value.

Usage Considerations

- Using a Data Location as the *count* Parameter

  The data location you use to specify the count parameter must evaluate to a nonnegative integer value; otherwise, Inspect displays this error message:

```
** Inspect error 183 ** Invalid length specified for DISPLAY
```

This example shows how the FOR clause with a variable used for the count affects the DISPLAY command output.

```
-OTEST1-DISPLAY string_ptr
STRING_PTR = "H"
-OTEST1-DISPLAY string_ptr FOR string_blen
STRING_PTR = "Hello World"
```

- Using FOR with FORMAT and PIC

  When you use the FOR clause with the FORMAT or PIC clauses, the FOR clause determines how much data to display and the FORMAT or PIC clause determines how to format the data.

Examples

The FOR clause examples are based on this TAL source code.

```
?SYMBOLS,NOLIST

STRUCT struct^def (*);
  BEGIN
  INT     i;
  STRING  s[0:29];
  END;

STRUCT struct^x ( struct^def );

PROC example MAIN;
BEGIN
  STRING .ptr;
  INT     str^blen;
  STRING .str[0:25];
  INT    .i^arr[0:10];

  struct^x.i ':=' 16 * [ 0 ];
  struct^x.s ':=' "This is example number one" -> @ptr;
  struct^x.i := @ptr '-' @struct^x.s;

  i^arr ':=' [ 0, 2, 6, 14, 30, 62, 126, 254, 510, 1022, 2046 ];
  str ':=' "Example text number two" -> @ptr;
  str^blen := @ptr - @str;
END;
```

1. This example displays *`struct^x`*, *`i^arr`*, *`str`*, and *`str^blen`* without any special formatting.

```
-PROGRAM-DISPLAY struct^x
STRUCT^X =
  I = 26
  S[0] = "This is example number one" ?0 ?0 ?0 ?0
-PROGRAM-DISPLAY i^arr
I^ARR[0] = 0 2 6 14 30 62 126 254 510 1022 2046
-PROGRAM-DISPLAY str,str^blen
STR[0] = "Example text number two" ?0 ?0 ?0, STR^BLEN = 23
```

2. This example displays the first four doublewords of *`struct^x`*.

```
-PROGRAM-DISPLAY struct^x FOR 4 DOUBLES
STRUCT^X = 1725544 1769152617 1931502968 1634562156
```

3. This example displays the first eight words of *`i^arr`* in octal (note that % appears only if the value exceeds eight).

```
-PROGRAM-DISPLAY i^arr FOR 2 QUADS IN OCTAL
I^ARR[0] = 0 2 6 %16 %36 %76 %176 %376
```

4. This example displays eight bytes of *str* in hexadecimal.

```
-PROGRAM-DISPLAY str FOR 8 IN HEX
STR[0] = %H45 %H78 %H61 %H6D %H70 %H6C %H65 %H20
```

5. This example displays the first 33 (value of *str^blen*) bytes of str.

```
-PROGRAM-DISPLAY str FOR str^blen
STR[0] = "Example text number two"
```

6. This example displays the first 10 words of *struct^x* in ASCII (note that “-” is the ASCII equivalent of 45, which is the value of *struct^x.i*).

```
-PROGRAM-DISPLAY struct^x FOR 10 WORDS IN ASCII
STRUCT^X = ?0 ?26 "This is example nu"
```

7. This example displays the first 45 (value of sstruct^x.i) bytes of *struct^x*.

```
-PROGRAM-DISPLAY struct^x FOR struct^x.i BYTES
STRUCT^X = ?0 ?26 "This is example number o"
```

Using the FORMAT Clause

The FORMAT clause directs Inspect to format a list of items using the operating system formatter.

```
DISPLAY item [ , item ]... { FORMAT | FMT } format-list


item:   one of

   display-data [ WHOLE ] [ PLAIN ] [ FOR group-spec ]
   display-code [ WHOLE ] [ PLAIN ] [ FOR group-spec ]
   REGISTER register-item [ TYPE display-type ]
   spi-buffer
   spi-token [ AS data-type ] [ FOR count ]
   string
   ( expression )
   VALUE value-list [ TYPE display-type ]


format-list:


         an edit-descriptor list for the operating system formatter
```

item [, *item*]

specifies the list of items you want to display.

`{ FORMAT | FMT }` *`format-list`*

directs Inspect to format items using the operating system formatter. format-list is a list of edit descriptors (with optional modifiers, decorations, and so on), separated by commas. The edit descriptors available in HP FORTRAN are also available in Inspect.

The operating system formatter provides several additional descriptors. For more information, see the *Guardian Programmer's Guide*.

Usage Considerations

- Size Limits on DISPLAY FORMAT Formatting

  Displays using the FORMAT clause cannot exceed one screen (24 lines) or 3168 bytes of data. Exceeding these limits causes Inspect to report an error instead of formatting and displaying the data.

- Using DISPLAY FORMAT in Command Lists

  If you use DISPLAY FORMAT in a command list, it must be the last command in the list because Inspect interprets all text following the FORMAT (or FMT) keyword as the format list.

- Using FORMAT with AS

  When you use the FORMAT clause with the AS clause, the AS clause determines how much data to display and the FORMAT clause determines how to format the data.

- Using FORMAT with FOR

  When you use the FORMAT clause with the FOR clause, the FOR clause determines how much data to display and the FORMAT clause determines how to format the data.

- Using FORMAT with TYPE

  You cannot use the FORMAT clause with the TYPE clause. If the TYPE clause precedes the FORMAT clause, the TYPE clause is ignored. If the TYPE clause follows the FORMAT clause, it is assumed to be part of the FORMAT clause text and will most likely yield an invalid FORMAT clause.

Examples

The variables used in the DISPLAY commands were declared as shown in this FORTRAN source program fragment:

```
       INTEGER i(8)
       REAL    x(8)
       DOUBLE PRECISION d(8)
       LOGICAL ERROR
       CHARACTER*40 stringvar
```

1. In this example, the integer array *I* is displayed. The second instance uses the FORMAT clause to display *I* as eight seven-digit integers using the *I* edit descriptor.

```
-FOROBJ-DISPLAY i
I[1] = 1 2 3 4 5 6 7 8
-FOROBJ-DISPLAY i FORMAT (8I7)
      1      2      3      4      5      6      7      8
```

2. In this example, the real array X is displayed. The second instance uses the FORMAT clause with the repeatable edit descriptors E, F and G to display X as various ten-digit fields.

```
-FOROBJ-DISPLAY x
X[1] =  3.1416  6.2832  9.424801  12.5664  15.708  18.8496  21.9912
25.1328
-FOROBJ-DISPLAY x FORMAT (E10.4,E10.2E2,F10.4,G10.4)
0.3142E+01  0.63E+01    9.4248 12.57
0.1571E+02  0.19E+02   21.9912 25.13
```

3. In this example, the real array D is displayed. The second instance uses the FORMAT clause with the repeatable edit descriptor D. The third instance shows the effect of the FOR clause when used in combination with the FORMAT clause.

```
-FOROBJ-DISPLAY d
D[1] =  9.8696517944335936  39.478607177734376  88.826873779296876
 157.9144287109375  246.74127197265625  355.3074951171875  483.61285400390626
 631.65771484375
-FOROBJ-DISPLAY d FOR 8 FORMAT (D20.12)
  0.986965179443E+01
  0.394786071777E+02
  0.888268737793E+02
  0.157914428711E+03
  0.246741271973E+03
  0.355307495117E+03
  0.483612854004E+03
  0.631657714844E+03
-FOROBJ-DISPLAY d[2] FOR 4 FORMAT (D20.12)  --  Start at second element.
  0.394786071777E+02
  0.888268737793E+02
  0.157914428711E+03
  0.246741271973E+03
```

4. This example shows the logical variable ERROR is displayed using the L edit descriptor.

```
-FOROBJ-DISPLAY error FORMAT (L3)
  F
```

5. This example shows the character string STRINGVAR is displayed. The second instance uses the A edit descriptor. The third instance adds the characters “e!” to the output being displayed.

```
-FOROBJ-DISPLAY stringvar
STRINGVAR = "This is a forty character string variabl"
-FOROBJ-DISPLAY stringvar FORMAT (A40)
This is a forty character string variabl
-FOROBJ-DISPLAY stringvar FORMAT (A40,'e!')  --  complete the string!
This is a forty character string variable!
```

6. This example shows how the FORMAT clause can be used with the VALUE clause to display an integer and a real value.

```
-FOROBJ-DISPLAY VALUE 10 FORMAT (I4)
  10
-FOROBJ-DISPLAY VALUE 25.678 FORMAT (E10.4)
0.2568E+05
```

Using the IN Clause

The IN clause directs Inspect to display a list of items in one or more bases.

```
DISPLAY item [ , item ]... IN base [ base ]...


item:   one of

   display-data [ WHOLE ] [ PLAIN ] [ FOR for-spec ]
   display-code [ WHOLE ] [ PLAIN ] [ FOR for-spec ]
   REGISTER register-item [ TYPE display-type ]
   spi-buffer
   spi-token [ AS data-type ] [ FOR for-count ]
   string
   ( expression )
   VALUE value-list [ TYPE display-type ]


base:   one of
   BINARY    OCT[AL]     DEC[IMAL]   HEX[ADECIMAL]
   ASCII     XASCII      GRAPHIC[S]
   ICODE
```

item [, *item*]...

specifies the list of items you want to display.

IN *base* [*base*]...

directs Inspect to display each item in one or more of these eight bases:

- ;BINARY displays items as binary values.
- OCTAL displays items as octal values.

- DECIMAL displays items as decimal values.
- HEXADECIMAL displays items as hexadecimal values.
- ASCII displays items as printable 7-bit ASCII characters. Inspect represents unprintable characters as a question mark followed by the character's numeric value expressed in the current output radix. For the ASCII base, unprintable characters are those with a decimal value less than 32 or greater than 126.
- XASCII displays items as printable 8-bit extended ASCII characters. Inspect represents unprintable characters as a question mark followed by the character's numeric value expressed in the current output radix. For the XASCII base, unprintable characters are those with a decimal value less than 32.
- GRAPHICS displays items as characters, including control characters.
- ICODE displays items as TNS instruction mnemonics, so it is invalid for PATHWAY requester programs.

Usage Considerations

- Displaying Items in Different Bases

  Because the IN clause applies to all the items listed, a single DISPLAY command cannot display one item using one base and another item using another base. This restriction can surface when you attempt to display code in ICODE and data in some other base. You can achieve this only indirectly, by displaying both items in both bases; for example:

```
-PRG-DISPLAY #main.label, #main.array IN ICODE ASCII
```

- Using IN for Data Conversion

  The IN format clause can be used for quick data conversion:

```
-PRG-DISPLAY "A" IN OCTAL
 %101
-PRG-DISPLAY (%167) IN ASCII
?0 "w"
```

Examples

1. This example illustrates displaying an integer array in octal.

```
-PROGRAM-DISPLAY three_var IN octal
THREE_VAR[0] = %30071 5 %173 2 %1605 %131 %1067 0
```

2. This example illustrates displaying the same array in binary.

```
-PROGRAM-DISPLAY three_var IN binary
THREE_VAR[0] = 0011000000111001 0000000000000101 0000000001111011
0000000000000010 0000001110000101 0000000001011001 0000001000110111
0000000000000000
```

3. This example illustrates using the IN clause with the FOR clause.

```
-PROGRAM-DISPLAY string_ptr IN a FOR 5
STRING_PTR = "Hello"
```

4. This TAL source code was used to generate this example.

```
PROC CODELOC;
BEGIN

  INT VARONE = 'P' := "LOCAL P RELATIVE ARRAY"; ! 11 WORDS
  INT VARTWO,I;

END;
```

5. This example illustrates using the FOR clause with the IN clause to display a code-location in both ASCII and ICODE.

```
-PROGRAM-DISPLAY #CODELOC FOR 4 IN ASCII
CODELOC = "LOCAL P "
-PROGRAM-DISPLAY #CODELOC FOR 4 IN ICODE
CODELOC =   STOR  G+117,6     LOAD  L+101,5     STOR  G+040,6     LDB  G+040
```

6. This TAL declarations apply to this example.

```
  INT           INTARRAY[0:1]   :=   [%H0,%H1234];

  STRUCT       .STRUCTARRAY[-1:1];
  BEGIN

    STRING       S[0:1];
    INT          IR = S;

  END;
```

7. This example illustrates displaying an element of an array in HEX.

```
-PROGRAM-DISPLAY INTARRAY[1] IN HEX
INTARRAY[1] = %H1234
```

8. This example illustrates displaying a structure member in DECIMAL.

```
-PROGRAM-MODIFY STRUCTARRAY[1].S[0] := 4
-PROGRAM-MODIFY STRUCTARRAY[1].S[0] := 5

-PROGRAM-DISPLAY STRUCTARRAY[1].IR  WHOLE  IN DECIMAL
STRUCTARRAY[1].IR = 1029
```

Using the PIC Clause

The PIC clause directs Inspect to format a list of items using mask strings and the M edit descriptor of the operating system formatter.

The M edit descriptor provides a formatting mechanism called “mask formatting,” which is similar to the COBOL PICTURE clause. For more information, consult the formatter documentation in the *Guardian Programmer’s Guide.*

```
DISPLAY item [ , item ]... PIC mask-str [ , mask-str ]...


item:   one of

   display-data [ WHOLE ] [ PLAIN ] [ FOR for-spec ]
   display-code [ WHOLE ] [ PLAIN ] [ FOR for-spec ]
   REGISTER register-item [ TYPE display-type ]
   spi-buffer
   spi-token [ AS data-type ] [ FOR for-count ]
   string
   ( expression )
   VALUE value-list [ TYPE display-type ]


mask-str:


         a mask string for the M edit descriptor
```

`item [ , item ]...`

specifies the list of items you want to display.

`PIC mask-str [ , mask-str ]...`

directs Inspect to format items using mask strings and the Edit descriptor of the operating system formatter. These characters have special functions in `mask-string` :

V	(uppercase V only)
Z	(uppercase Z only)
9	
.	

The formatter displays all other characters (including lowercase z and v) exactly as they appear in `mask-string`. Mask strings and the M edit descriptor are described fully in the *Guardian Programmer’s Guide*.

Usage Considerations

- Size Limits on DISPLAY PIC Formatting

  Displays using the PIC clause cannot exceed one screen (24 lines) or 3168 bytes of data. If these limits have been exceeded, the data is displayed and Inspect issues this warning message:

```
** Inspect warning 87 ** Maximum lines exceeded for format output
```

- Differences between PIC and PICTURE

  Note that there are significant differences between the INSPECT PIC clause and the COBOL PICTURE clause. The PICTURE clause is part of a data item's definition, while the PIC clause is simply a template for formatting a data item. Also, the PIC clause is case-sensitive, excludes constructs such as parenthesized repetition counts, and does not perform floating replacement.

- Using PIC with AS

  When you use the PIC clause with the AS clause, the AS clause determines how much data to display and the PIC clause determines how to format the data.

- Using PIC with FOR

  When you use the PIC clause with the FOR clause, the FOR clause determines how much data to display and the PIC clause determines how to format the data.

- Using PIC with TYPE

  You cannot use the PIC clause with the TYPE clause.

Examples

This source code was used to generate the examples for the PIC clause.

```
?INSPECT, SYMBOLS

        IDENTIFICATION DIVISION.
        PROGRAM-ID.  example.
        DATA DIVISION.
        WORKING-STORAGE SECTION.
        01    var1          PIC 99 COMP VALUE 2.
        01    var2          PIC X(8).
        01    arr1          GLOBAL.
              02  sub1-arr  PIC XXXXX   VALUE "12345".
              02  sub2-arr  PIC X       VALUE "5".
              02  sub3-arr  PIC XXX     VALUE "123".
              02  sub4-arr  PIC XX      VALUE "2".

        PROCEDURE DIVISION.
        exampletoshow.
            MOVE "12.34" TO var2.
            DISPLAY "Hello World".
             STOP RUN.
```

1. This example displays sub-elements of *ARR1* using the PIC clause.

```
-PROGRAM-DISPLAY arr1 PIC "ZZZ999"
 12345
   005
   123
  02
```

2. This example displays var1 and arr1 with decimal alignment as stated in the PIC clause.

```
-PROGRAM-DISPLAY var1 PIC "ZZ99.99"
  02.00
-PROGRAM-DISPLAY arr1 PIC "ZZZ.99"
 123.45
     05
   1.23
     2
```

3. This example displays arr1 with implied decimal alignment as stated in the PIC clause.

```
-PROGRAM-DISPLAY arr1 PIC "ZZZZZV99"
  12345
     05
    123
     2
```

4. This example shows the usage of the PIC clause of the DISPLAY command with the FOR clause.

```
-PROGRAM-DISPLAY arr1 FOR 1 PIC "ZZ.99"
   01
-PROGRAM-DISPLAY arr1 FOR 5 PIC "ZZ99"
  01
  02
  03
  04
  05
```

Using the TYPE Clause

The TYPE clause directs Inspect to display an item using the attributes of a display type defined by Inspect.

Note. The TYPE clause can also direct Inspect to display an SPI buffer or token. This second usage of the TYPE clause is described in the subsection “Displaying SPI Data.”

```
DISPLAY item [ TYPE display-type ]


item:   one of

   data-location
   REGISTER register-item
   VALUE value-list


display-type:  one of

   CRTPID       DEVICE       ENV                  FILENAME
   FILENAME32   LOCATION     PROCESS HANDLE       SSID
   SYSTEM       TIMESTAMP    TIMESTAMP48          TOSVERSION
   TRANSID      USERID       USERNAME
```

item

specifies the item you want to display. You can specify a data address, a program register, or a list of values as the item.

TYPE *display-type*

directs Inspect to display the item using the attributes of the given display type.

CRTPID

interprets the item as an 8-byte process identifier and displays it in this form:

```
[\system.][$process-name] cpu,pin
```

For more information on CRTPIDs, see the *Guardian Programmer’s Guide*.

DEVICE

interprets the item as an 8-byte device name and displays it in this form:

```
[\system.]$device-name
```

ENV

interprets the item as a 16-bit environment register and displays it in this form:

```
(RP=rp,condition-code[,flag]...)  space-id
```

FILENAME

interprets the item as a 24-byte internal file name and displays it in one of these forms:

```
[\system.]$volume.subvolume.filename
[\system.]$volume.#number
```

The first form represents a permanent file and the second form represents a temporary file.

`FILENAME32`

interprets the item as a 32-byte internal file name and displays it in one of these forms:

```
\system.$volume.subvolume.filename
\system.$volume.#number
```

The first form represents a permanent file and the second form represents a temporary file.

`LOCATION`

interprets the item as a 1word code address and displays it using all location formats: STATEMENTS OFFSET, LINES FILE ALL OFFSET, and INSTRUCTIONS.

You can specify that the address is from a different code segment by entering a segment identifier after the LOCATION display type. The segment identifier has this form:

```
{ UC | UL } [ .segment-number ]
```

UC indicates that the address is in the user code space; UL indicates that it is in the user library space. *`segment-number`* specifies the particular code segment within the code space; it must be a number in the range zero through 31. If you omit the segment number, Inspect uses the value zero.

`PROCESS HANDLE`

interprets the item as an 10-word process identifier and displays it in this form:

```
\system.[$process-name:]cpu:pin:verification-sequence-
number
```

For more information on process handles, see the *Guardian Programmer's Guide.* Note that process handle may not work if the current program is a savefile because it is not active.

`SSID`

interprets the item as a 12-byte subsystem identifier and displays it in one of these forms:

```
owner.subsys-name.version
owner.subsys-number.version
```

`SYSTEM`

interprets the item as a 2-byte field and displays it in this form:

```
\system
```

`TIMESTAMP`

interprets the item as a 64-bit timestamp and displays it in one of these forms:

`year-month-day hour``:`*`min`*`:`*`sec`*`:`*`millisec`*`:`*`microsec`*
`n`` days, `*`n`*` hours, `*`n`*` min, `*`n`*` sec, `*`n`*` millisec, `*`n`*` microsec`

The first form represents a date and time and the second form represents elapsed time. Inspect uses elapsed time if the year is less than one.

`TIMESTAMP48`

interprets the item as a 48-bit timestamp and displays it in this form:

`year-month-day hour:min:sec:centisec`

`TOSVERSION`

interprets the item as a 2-byte operating system version identifier and displays it in this form:

`R`*`nn`*

`TRANSID`

interprets the item as an 8-byte TMF transaction identifier and displays it in the form defined by the TRANSIDTOTEXT system procedure.

`USERID`

interprets the item as a 2byte user identifier and displays it in this form:

`group-number,user-number`` [`*`group-name.user-name`*`]`

Inspect includes *`group-name.user-name`* if the given user identifier is defined on the system hosting the Inspect process.

`USERNAME`

interprets the item as a 16-byte user name and displays it in this form:

`group-name.user-name`` [`*`group-number,user-number`*`]`

Inspect includes *`group-number,user-number`* if the given user name is defined on the system hosting the Inspect process.

Usage Considerations

- Using TYPE with Save files

  When you use the TYPE clause to display data from a save file, Inspect uses the local system and current network to expand system names and user names. If the save file was created on another system or another network, Inspect might report incorrect or undefined system names and user names.

- Using TYPE with REGISTER

  When you use the TYPE clause with the REGISTER clause, you can only specify display types that represent 2byte values.

- Using TYPE with VALUE

  When you use the TYPE clause with the VALUE clause, you must ensure that the size of the values you provide matches the size of the display type you specify.

  Display types CRTPID, DEVICE, FILENAME, FILENAME32, PROCESS HANDLE, SSID, and USERNAME may not be used with VALUE.

- Using TYPE with FOR

  When you use the TYPE clause with the FOR clause, the display type in the TYPE clause overrides the unit in the FOR clause.

- Using TYPE with FORMAT and PIC

  You cannot use the TYPE clause with the FORMAT or PIC clauses.

- Using TYPE with IN

  When you use the TYPE clause with the IN clause, the TYPE clause determines how much data to display and the IN clause determines how to format the data. In addition, the IN clause cannot be used with the TYPE clause of the DISPLAY command.

Examples

1. This example displays a TYPE CRTPID.

```
-PROGRAM-DISPLAY variable TYPE CRTPID
$Y667 11,53
-PROGRAM-DISPLAY variable TYPE CRTPID FOR 2
$Y667 11,53
$Z534 8,100
```

2. This example displays a TYPE DEVICE.

```
-PROGRAM-DISPLAY variable TYPE DEVICE
$DISK
```

3. This example displays a TYPE ENV.

```
-PROGRAM-DISPLAY variable TYPE ENV
(RP=7,CCG,T)   UC.7
```

4. This example displays a TYPE FILENAME.

```
-PROGRAM-DISPLAY variable TYPE FILENAME
$DATA.SUBVOL.FILE
```

5. This example displays a TYPE FILENAME32.

```
-PROGRAM-DISPLAY variable TYPE FILENAME32
\SYS.$DISK.SUBVOL.FILENAME
```

6. This example displays a TYPE LOCATION.

```
-PROGRAM-DISPLAY variable TYPE LOCATION
#TEST^PROC.7 + %3I, #TEST^PROC.#223.2(\SYS.$DATA.SUBVOL.FILENAME) + %3I,
#TEST^PROC +
%1030I
-PROGRAM-DISPLAY variable TYPE LOCATION UC.3
#READ^INFO.15 + %5I, #READ.INFO.#553(\SYS.$DATA.SUBVOL.FILENAME) + %5I,
#READ^INFO +
%756I
```

7. This example displays a TYPE PROCESS HANDLE. A process handle can be for a named or unnamed process. The first example shows the format Inspect uses for a named process; the second shows the format Inspect uses for an unnamed process.

```
-PROGRAM-DISPLAY variable TYPE PROCESS HANDLE
\SYS.$Z365:3:60:10352657
-PROG-DISPLAY variable2 TYPE PROCESS HANDLE
\SYS.4:42:10931482
```

8. This example displays a TYPE SSID.

```
-PROGRAM-DISPLAY variable TYPE SSID
TANDEM.EMS.C00
```

9. This example displays a TYPE SYSTEM.

```
-PROG-DISPLAY variable TYPE SYSTEM
\SYS
```

10. This example displays a TYPE TIMESTAMP.

```
-PROGRAM-DISPLAY variable TYPE TIMESTAMP
1987-09-30 19:47:14.072.397
```

11. This example displays a TYPE TIMESTAMP48.

```
-PROGRAM-DISPLAY variable TYPE TIMESTAMP48
2002-05-04 18:39:01.76
```

12. This example displays a TYPE TOSVERSION.

```
-PROGRAM-DISPLAY variable TYPE TOSVERSION
D10
```

13. This example displays a TYPE TRANSID.

```
-PROGRAM-DISPLAY variable TYPE TRANSID
\SYS.14.904
```

14. This example displays a TYPE USERID.

```
-PROGRAM-DISPLAY variable TYPE USERID
255,255 SUPER.SUPER
```

15. This example displays a TYPE USERNAME.

```
-PROGRAM-DISPLAY variable TYPE USERNAME
SUPER.SUPER 255,255
```

ENV

The ENV command displays the current settings of the Inspect environment and selectable parameters.

```
ENV [ env-parameter ]

env-parameter:   one of

   DIRECTORY
   LANGUAGE
   LOG
   PROGRAM
   SCOPE
   SOURCE SYSTEM
   SYSTEM
   SYSTYPE
   VOLUME
```

Default Value

If no options are specified, Inspect displays the values for all parameters.

Usage Considerations

- Establishing a new default system

  When ENV displays blanks after the word SYSTEM, it means that you have not used the Inspect command SYSTEM to establish a new default system. Inspect will use the system it was installed on.

- Valid systypes

  Inspect’s systype can be either Guardian or OSS.

- Default Volume and Subvolume

  If an Inspect session starts as a result of a debug event—for example, RUND PRG—Inspect uses your logon volume and subvolume as the default volume and subvolume for the session. If you start an Inspect session using the TACL command RUN INSPECT, then Inspect uses your current volume and subvolume as the default volume and subvolume.

- SOURCE SYSTEM and SYSTEM

  Inspect uses both SOURCE SYSTEM and SYSTEM to expand partially qualified file names. Inspect uses the current SOURCE SYSTEM value to expand source file names, and it uses the current SYSTEM value to expand all other file names.

Related Commands

- LOG on page 6-142
- PROGRAM on page 6-156
- SCOPE on page 6-162
- SELECT LANGUAGE on page 6-166
- SELECT PROGRAM on page 6-167
- SELECT SOURCE SYSTEM on page 6-169
- SOURCE SYSTEM on page 6-211
- SYSTEM on page 6-217
- VOLUME on page 6-223

EXIT

The EXIT command stops the Inspect process. Programs being debugged with Inspect are not stopped.

```
EXIT
```

Usage Considerations

- Pressing CTRL/Y is the same as entering EXIT.
- Entering an EXIT command does not alter any of the programs you are debugging. Consequently, you should stop all programs or clear all breakpoints from all programs and RESUME them before you enter an EXIT command.
- Use the TACL commands ACTIVATE and STOP to control a suspended process. Use the PATHCOM command ABORTþTERM to stop suspended PATHWAY terminals.

- If you issue an EXIT command interactively and one or more active processes are running under the control of Inspect, Inspect displays a warning message. Inspect emits a second warning message if you have any breakpoints set. For example:

```
-PROGRAM-EXIT
WARNING - You have 2 active processes:
             Program
Num PID      Name       Type         State Location
  1 01,00307 X003OBJ1   TNS          HOLD  #X001A.#27.4(X003TAL) + %2I
 *2 09,00282 X003OBJ0   TNS          HOLD  #main.#8(X003SRC)
WARNING - The processes above will not be stopped.
```

  Entering “Y” at the confirmation prompt will exit Inspect, but will not stop the processes under the control of Inspect. All processes will remain under the control of Inspect in their current state. If you do not enter a Y, Inspect does not exit; instead, it issues a prompt. If you respond “Yes” when prompted for exit confirmation, you can reactivate your process with the TACL ACTIVATE command.

- If you are running a PATHWAY requester program and you type EXIT, you will not be prompted to confirm that you want to stop Inspect unless you have breakpoints set.

Related Command

RESUME with the * EXIT option

FA

The FA (“Fix Alias”) command enables you to retrieve, edit, and reissue an existing alias definition.

When you enter an FA command, Inspect presents the ADD ALIAS command for the specified alias in an editing template. For more information, see Editing Templates on page 6-85.

```
FA alias-name
```

alias-name

  specifies the name of the alias you want to fix.

Usage Consideration

Aliases are not expanded when you use the FA command.

Related Commands

- ADD ALIAS on page 6-7
- ALIAS on page 6-17

- LIST ALIAS on page 6-131

Example

This example illustrates the FA command.

```
-PRG-FA ShowVars
-ADD ALIAS SHOWVARS = "DISPLAY height, width, depth"
.          ddddiWatch//                            i; RESUME
-ADD ALIAS WatchVARS = "DISPLAY height, width, depth; RESUME"
.
```

FB

The FB (“Fix Breakpoint”) command enables you to retrieve, edit, and reissue an existing breakpoint definition.

When you enter an FB command, Inspect presents the BREAK command for the specified breakpoint in an editing template (for code breakpoints only). For more information, see Editing Templates on page 6-85.

```
FB breakpoint-number
```

breakpoint-number

specifies the number (as shown by the LIST BREAKPOINT command) of the breakpoint to fix.

Usage Considerations

- You cannot use the FB command with data breakpoints or breakpoints created in Debug or a previous Inspect session.
- You can only edit code breakpoints that have an ordinal number. The maximum ordinal number is 99; therefore, you cannot edit code breakpoints listed after breakpoint 99.
- Aliases are not expanded with the FB command.

Related Commands

- BREAK on page 6-19
- LIST BREAKPOINT on page 6-131

FC

The FC (“Fix Command”) command enables you to retrieve, edit, and reissue a command line in the history buffer.

When you enter an FC command, Inspect presents the specified command line (up to 255 characters) in an editing template. Inspect will issue a warning if you exceed 255 characters on an insert or a replace.

For more information, see Editing Templates on page 6-85.

```
FC [ command-line-specifier ]


command-line-specifier:   one of

   pos-num
   neg-num
   search-text
   "search-text"
```

cmd-line-specifier

specifies which command line to retrieve from the history buffer.

pos-num

is a positive integer that refers to the command-line number in the history buffer that you want to retrieve.

neg-num

is a negative number that refers to a command line in the history buffer relative to the current command line.

search-text

is the most recent command line in the history buffer that begins with the text you specify. You need to specify only as many characters as necessary to identify the command line uniquely.

"*search-text*"

is similar to search-text except it the text may appear anywhere in a line.

Default Value

If you do not specify a command line, Inspect retrieves the last command line you entered, excluding FA, FB, FC, FK, XC, and “!”.

Editing Templates

When you use one of the Fix commands—FA, FB, FC, or FK—Inspect retrieves a command line and presents it in an editing template. You can then alter and reissue the command or command line. A editing template has the form:

```
-current template contents
.
```

The first line is the contents line; it displays the text that you are editing. The second line is the editing line; here, you enter replacement text and editing commands.

Note that the contents line in the preceding example begins with a dash, which is the command-mode indicator. A dash indicates high-level Inspect and an underscore indicates low-level Inspect. When you reissue the command or command line that you are editing, Inspect uses the current command mode to interpret it.

On the editing line, use the space bar and the backspace key to position the cursor under the text in the contents line that you want to change. Do not use the arrow keys to move the cursor.

After you make changes, press the RETURN key. Inspect then redisplays the editing template; the contents line reflects the changes you have made. You can add more changes at this point. However, if the command is correct, press the RETURN key again, and Inspect executes the command. If you want to terminate editing without reissuing the edited command, enter two slashes at the start of the editing line and then press RETURN. Alternatively, you can press CTRL/Y or BREAK at any time to terminate editing.

Using the Editing Characters D, I, and R

On the editing line you can use three editing commands:

D or d	Deletes the character above the D
I or i	Inserts the text following the I into the command line; text is inserted in front of the character above the I.
R or r	Replaces characters in the command line (beginning with the character above the R) with the text following the R.

You must begin your correction with these editing characters if the first character of the change is I, D, or R. Type the D or R under the character to be deleted or replaced. Type the I under the character that follows the insert position.

Spaces typed after the I or R command are part of the text to insert or replace. If you want to make more than one change on a line, end the text string with two slashes (//) and space over to make additional changes; for example:

```
-PRG-FA ShowVars
-ADD ALIAS SHOWVARS = "DISPLAY height, width, depth"
.          ddddiWatch//                             i; RESUME
-ADD ALIAS WatchVARS = "DISPLAY height, width, depth; RESUME"
.
```

In the example, the D commands delete “SHOW”; the first I command inserts a “Watch” before “VARS.” The two slashes indicate the end of this insertion. The second I command inserts the string “; RESUME” before the quotation mark that terminates the alias's replacement string. After you press the RETURN key, Inspect redisplays the editing template and prompts you for any further changes. Because the command is now correct, make no changes before you press RETURN; Inspect then executes the modified command.

Replacing and Inserting Text without Using D, I, and R

You can enter a replacement string on the editing line without using the R command. You can also insert text at the end of the line by typing it in without using the I command; for example:

```
-PRG-FC
-D currec.next.
M
-M currec.next
               = 0
-M currec.next = 0
.
```

However, if your proposed replacement or inserted text begins with any of the letters D, I, or R (or their lowercase counterparts), Inspect considers that letter to be an editing command, and you do not get the result you want; for example:

```
-PRG-FC
-STEP 5 STATEMENTS
.       INSTRUCTIONS
-STEP 5 NSTRUCTIONSSTATEMENT
.
```

The intent of this Fix command was to change the code units in the STEP command from STATEMENTS to INSTRUCTIONS. However, Inspect read the I in “INSTRUCTIONS” as the I command, and so inserted “NSTRUCTIONS” in front of “STATEMENTS” instead. At this point, rather than trying to execute or edit the command, you can enter two slashes and press the RETURN key. The FC command terminates and Inspect prompts for a command. All changes to the line are abandoned.

Usage Consideration

Aliases are not expanded with the FC command.

Related Commands

- HISTORY on page 6-93
- LIST HISTORY on page 6-135
- XC on page 6-224

Example

This example illustrates the FC command:

```
--ALIAS mult ="display (5*5)"
 --key f1 = "resume"
 --set radix output octal
 --history
1: ALIAS mult ="display (5*5)"
2: KEY F1 = "resume"
3: SET RADIX output octal
--fc 1
-ALIAS mult ="DISPLAY (5*5)".
                        6
-ALIAS mult ="DISPLAY (5*6)"
.
--FC -2
-SET RADIX output octal.
                 rdecimal
-SET RADIX output decimal
.
--HISTORY
1: ALIAS mult ="display (5*5)"
2: KEY f1 = "resume"
3: SET RADIX output octal
4: ALIAS mult ="display (5*6)"
5: SET RADIX output decimal
--FC ali
-ALIAS mult ="display (5*6)"
.                          i in octal
-ALIAS mult ="display (5*6) in octal"
.
--FC "f1"
-KEY f1 = "resume"
.          imult;
-KEY f1 = "mult;resume"
.
--
```

FILES

The FILES command shows the status of files that have been opened by the current program. The FILES command is a synonym for the INFO OPENS command.

Note. The FILES command is invalid for PATHWAY requester programs.

```
FILES [ { * | file-list } [ DETAIL ] [ file-type ]


file-list
   file-number  [, file-number  ]...


file-type:   one of

   FORTRAN   FD   GUARDIAN
```

`*`

requests the status of all files opened by the current program.

`file-list`

requests the status of specific files. *`file-list`* is a list of file numbers identifying the desired files.

`file-number`

specifies a single file. *`file-number`* can be:

A COBOL FD name
An expression that evaluates to an integer value
A data location identifying an integer value

Inspect interprets the integer values as file-system numbers, unless you specify the F clause; Inspect would then interpret the values as FORTRAN logical unit-numbers. If a FD clause is used, Inspect interprets the values as OSS file descriptors.

`DETAIL`

directs Inspect to display the maximum available information for the specified files.

`file-type`

indicates to Inspect what type of file to display. *`file-type`* can be one of the following:

`FORTRAN`

indicates that *`file-number`* values specify FORTRAN logical-unit numbers, not file system file numbers. Files without FORTRAN logical units will not be displayed.

The F clause is valid only in FORTRAN scope units.

`FD`

indicates that *`file-number`* values specify OSS file descriptors. Files without OSS file descriptors will not be displayed.

`GUARDIAN`

indicates that *`file-number`* values specify Guardian system file numbers. Files without Guardian file system numbers will not be displayed.

FK

The FK ("Fix Key") command enables you to retrieve, edit, and reissue an existing function-key definition.

When you enter an FK command, Inspect presents the ADD KEY command for the specified function-key definition in an editing template. For more information, see Editing Templates on page 6-85.

```
FK key-name


key-name:   one of

   F1     F2     F3     F4     F5     F6     F7     F8
   F9     F10    F11    F12    F13    F14    F15    F16
   SF1    SF2    SF3    SF4    SF5    SF6    SF7    SF8
   SF9    SF10   SF11   SF12   SF13   SF14   SF15   SF16
```

`key-name`

specifies the function key you want to fix. Valid function keys include F1 through F16 and shifted F1 (SF1) through shifted F16 (SF16).

Usage Consideration

Aliases are not expanded with the FK command.

Related Commands

- ADD KEY on page 6-9
- DELETE KEY on page 6-30
- KEY on page 6-128
- LIST KEY on page 6-136

HELP

The HELP command displays information about Inspect commands, features, and concepts. HELP text is organized in a hierarchical fashion; general topics at the top level, with more specific subtopics at each lower level. After displaying help for a topic, Inspect will display a list of subtopics, if any, and prompt you for the subtopic for which you desire information.

```
HELP [ topic ]


topic:

   main-topic [ sub-topic [ sub-topic ] ]
```

topic

is an Inspect help topic. A help topic consists of a main topic and up to two levels of subtopics. Aliases may not be used in topics or within HELP.

Default Value

If you enter the HELP command without specifying a topic, Inspect displays what main topics are valid for the current mode (high-level or low-level).

Usage Guidelines

- Help Topics

  The HELP command provides helpful information about various topics, including:

  - Inspect commands, both low-level and high-level
  - Language-dependent parameters such as code locations
  - Errors
  - Highlights from the current and earlier releases
  - Corrected and known problems

- Help Subtopics

  The HELP command provides subtopic options within topic descriptions. Subtopics might include:

  - Command descriptions
  - Syntax
  - Examples
  - Usage considerations
  - Related commands
  - Default values
  - Errors

    To receive information on a specific error, at the first-level Inspect HELP prompt enter:

    ```
    Error num
    ```

  - More Help ([Y]/N)? prompt

    When you request help for a topic, Inspect displays information for the topic. If the information exceeds 23 lines, Inspect displays the first 23 lines and then prompts you:

    ```
    More Help ([Y]/N)?
    ```

    Enter Y or a carriage return to continue the display. Entering "N" will end the current display and back you out of the current level of help.

  - Using Online HELP from within HELP

    After displaying the information for a topic, Inspect prompts you for a subtopic, if any exist for the given topic. You can then select a subtopic or press RETURN to exit HELP and return to the Inspect prompt. You can exit HELP and return to the Inspect prompt by pressing CTRL/Y at any help prompt.

    When a list of topics is presented, you only need qualify the topic name with as many characters needed to provide uniqueness. If the number of characters does not uniquely qualify a topic, the first topic in the list that matches the input will be chosen.

  - Using Online HELP from the Inspect command line

    You will achieve better results if you completely spell out the topic and/or the subtopic name. If, for example, you want help on INFO OPEN, do not type "HELP I O", you will receive help on INFO OBJECTFILE instead of INFO OPEN. Similarly, "HELP C" will return help on the CLEAR command instead of the C language, because Inspect interprets C as an abbreviation for the CLEAR command.

HISTORY

The HISTORY command displays the most recently executed command lines.

```
HISTORY  [ num ]
```

num

is the number of commands to display.

Default Value

If no number is specified, the last 10 commands are displayed.

Usage Consideration

The HISTORY command is equivalent to LIST HISTORY -num/-1, except when less than *num* command lines are in the history buffer, in which case all lines in the history buffer are displayed.

Related Commands

- FC on page 6-84
- LIST HISTORY on page 6-135
- XC on page 6-224

HOLD

The HOLD command suspends the execution of a program on the program list, placing it in the hold state. Inspect does not begin prompting again until all suspensions are accomplished.

```
HOLD [ program [ , program ]...]
     [ *                       ]

program:   one of

   program-number
   program-name
   cpu,pin
   Pathway-term-name
```

program

specifies a program using one of several formats. *program-number* identifies the program by its program number (as shown by the LIST PROGRAM command). *program-name* identifies the program by its program name (as shown by the LIST

PROGRAM command). `cpu,pin` identifies a process by its process ID. PATHWAY-term-name identifies a PATHWAY requester program by the name of its logical PATHWAY terminal.

`*`

specifies all executable programs in the program list.

Default Value

Entering HOLD without parameters suspends the current program.

Usage Considerations

- Holding a Program

  When a program enters the hold state, it becomes the current program. In the case of multiple suspensions, the last one suspended becomes the current program. To determine the current program, enter the LIST PROGRAM command; an asterisk precedes the program number of the current program. Use SELECT PROGRAM to change the current program.

- HOLD During a System Procedure Call

  If program execution is in a system procedure call, Inspect will wait until the system procedure returns control to the program before suspending execution. Therefore, when you suspend several programs, the order of suspension can be different from the order of the HOLD commands.

- Pressing the BREAK Key

  Pressing the BREAK key while the current program is running does not place the program in the hold state; you must use the HOLD command.

ICODE

The ICODE command displays instruction mnemonics starting at a specified statement or code address. For accelerated programs, the ICODE command may be used to list TNS instructions, TNS/R instructions, or a combination of both.

Note. The ICODE command is invalid for PATHWAY requester programs.

```
ICODE location [ FOR count [ unit ] ] [ report  ]

location:  one of

   [ AT display-code  ]
   [ tns/r tns/r-address-expression ]
   [ tns-address-expression ] [ UC.number | UL.number ]

unit:  one of

     INSTRUCTION[S]
     STATEMENT[S]
     VERB[S]

report:  one of

   TNS
   tns/r
   BOTH

tns/r:  one of

   TNSR
    TNS/R
    R

tns/r-address-expression:

   tns/r-value [ operator tns/r-value ]...

tns-address-expression:

   tns-value [ operator tns-value ]...

operator:   one of

   *   /   <<   >>   +   -

tns-value:   one of

   ( tns-expression )
   16-bit number
   tns-register

tns/r-value:   one of
   ( tns/r-address-expression )
   32-bit number
   16-bit number [ .16-bit number ]
   tns/r-register
```

location

specifies the location to begin listing instruction mnemonics.

`AT` *`display-code`*

specifies a source-level code location (such as a line number) starting at which instructions are to be listed.

`TNS/R` *`tns/r-address-expression`*

specifies the TNS/R machine address, for accelerated or native programs, starting at which instructions are to be listed.

`tns-address-expression` `[ UC.`*`number`* `| UL.`*`number`* `]`

specifies the TNS code address beginning where instructions are to be listed. The code segment number must be in the range zero through 31.

`count`

specifies the number of instruction units to be displayed.

`unit`

specifies whether the instructions to be listed are in the form of statements, instructions, or verbs.

`STATEMENTS`

specifies that instructions are to be listed in units of statements. (This is the default when the AT clause is used.)

`INSTRUCTIONS`

specifies that count is the number of instructions to be listed. (This is the default when an address expression is specified.)

`VERBS`

specifies that the instructions are to be listed in units of verbs (for COBOL and COBOL85 programs).

`report`

specifies what type of instructions are to be displayed.

`TNS`

instructs the ICODE command to list TNS instructions.

`TNS/R`

instructs the ICODE command to list TNS/R instructions.

`BOTH`

instructs the ICODE command to list the correspondence between TNS and TNS/R instructions for accelerated programs. Refer to the examples for more details. The BOTH option does not show TNS/E instructions for an OCA process on a TNS/E system.

Default Values

- INSTRUCTIONS are assumed if no unit is specified and an address expression is specified.
- STATEMENTS are assumed for non-COBOL programs if no unit is specified and a source-level location is specified using the AT clause.
- VERBS are assumed for COBOL and COBOL85 programs if no unit is specified and a source-level location is specified using the AT clause.
- A count of one is assumed if no count is specified.
- By default, this command lists TNS instructions, unless the BOTH or TNS/R options are specified.
- As with the SOURCE command, pressing RETURN immediately after an ICODE command has been issued will repeat the ICODE command, starting at the location following the last location listed.

Output

- TNS addresses are output in 5-digit octal with a leading “%”.
- TNS/R addresses are output in 8-digit hexadecimal with a leading “%h”.
- When listing TNS/R call instructions (JAL and JR) the name of the target procedure or millicode routine is listed.
- When a source location is specified, output includes line and/or statement numbers (depending on the current LOCATION FORMAT).
- The BOTH report lists the correspondence between TNS and TNS/R instructions in a vertical format. It lists a block of TNS instructions and the corresponding block of TNS/R instructions. Memory-exact points determine the block boundaries. In output, blocks are separated by a blank line or a line/statement number (if applicable). This output is illustrated in the examples section.
- For accelerated programs, instructions are annotated in the standard way to mark memory-exact and register-exact points, a “>” and an “@”, respectively.
- Since procedure names may be up to 255 characters in length, strings listed for some call instructions may overflow the allocated field width. If the string will fit on the current output line, it is allowed to overflow into subsequent fields as necessary. If not, it is output at the beginning of the next line and folded across as many lines as necessary.

Usage Considerations

- The AT clause and the STATEMENTS and VERBS units can only be used when symbol information is available.
- If the AT clause is used with a line number for which there is no corresponding statement, a warning is issued and the next line number for which there is a corresponding statement is assumed.
- When a code source-level location is specified, instructions cannot be listed past the end of the containing procedure.
- When an address is specified and STATEMENTS are specified as the unit, instructions are listed from the address to the end of the containing statement.

Usage Considerations for Accelerated Programs on TNS/E Systems

- The ICODE command shows safe-point annotations for an OCA process on a TNS/E system, but not for an OCA snapshot. For example, the > shown here is displayed for an OCA process (but does not appear for a snapshot of the same process):

  ```
  #36
    %000037:  >  LWP   +021       LBA      STB    L+001
  ```

  The ICODE command shows only TNS instructions for an accelerated program on the TNS/E platform. However, you can use Visual Inspect or TNSVU to view safepoints and corresponding TNS and TNS/E instructions in an OCA process or snapshot on a TNS/E system.
- The BOTH report does not show TNS/E instructions for OCA processes.

Usage Considerations for Accelerated Programs on TNS/R Systems

- Listings that depend on the mapping between TNS and TNS/R addresses begin at the previous memory-exact point if the given address is not a memory-exact point. This occurs in these cases:
  - When listing TNS instructions given a TNS/R address that is not a memory-exact point.
  - When listing TNS/R instructions given a TNS address that is not a memory-exact point.
  - When listing TNS/R instructions given a source-level location or unit. In such an event, this warning is reported:

```
** Inspect warning 372 ** Starting location is not a memory-exact point;
                          listing begins at preceding memory-exact point
```

- If the location at the end of the range is not an exact point, instructions are listed till the next exact point and this warning is issued:

```
** Inspect warning 387 ** Ending location is not a memory-exact point;
                          listing ends at following memory-exact point
```

- If a TNS/R code address that is not 32-bit aligned is input, it is rounded up to the next 32-bit boundary and this warning is issued:

```
** Inspect warning 389 ** Address is not aligned to a TNS/R instruction
                           boundary it will be rounded down to the previous
                           instruction
```

- The TNS/R and BOTH clauses cannot be used when debugging a program that has not been processed by the Axcel accelerator. An attempt to do so will result in this error:

```
** Inspect error 361 ** Operation available only on an accelerated program
```

- Use of the STATEMENTS unit with a TNS/R address will list the proper number of instructions only when the address corresponds to or is rounded to the beginning of the statement.

Related Command

SOURCE ICODE

Examples

1. These are the examples of possible ICODE commands.

   - To list four TNS instructions, starting three instructions before the current location:

     ```
     ICODE p - 3 FOR 4
     ```

   - To list two TNS/R instructions, starting one instruction before the current location:

     ```
     ICODE R $PC - 4 FOR 2
     ```

   - To list 10 TNS instructions, starting at the computed address:

     ```
     ICODE %2323 + %57 FOR 10
     ```

   - The Debug/CRUNCH/low-level Inspect dot operator is supported, allowing this TNS/R address notation to be used:

     ```
     ICODE %h70420.%h00AA
     ```

2. This illustrates listing TNS instructions starting at a given line for two statements. Note that pressing RETURN lists the ICODE of two additional statements.

```
-OTALIN- ICODE at #73 for 2s
#73
  %000052:    LDI    +001            ADDS   -002            RSUB    01
  %000055:    ADDS   +002            LDI    +010            LDI    +000
  %000060:    PUSH    711
#75.1
  %000061:    LDI    +001            STOR  G+000
-OTALIN-
#76
  %000063:    LDI    +001            STOR  L+001
#77
  %000065:    BSUB   -033
```

3. This illustrates listing TNS instructions starting at a given TNS address. Note that statement numbers are not listed when an address is specified.

```
-OTALIN-ICODE %65 for 20
  %000065:    BSUB   -033            LOAD  L+001            LADR  L+001
  %000070:    LOAD  L+001            PUSH    722            BSUB   -033
  %000073:    BSUB   -025            STRP    7              LOAD  L+001
  %000076:    STOR  L+002            LDI    +001            LDI    +000
  %000101:    LDI    +002            LDD   L+003            PUSH    744
  %000104:    PCAL    002            LDI    +001            LDI    +000
  %000107:    LDI    +002            LDD   L+003
```

4. This is an example of listing TNS/R instructions from an Accelerated program.

```
-PTALIN-ICODE r %h70420194 for 10
  %h70420194: > LI    s0,1          SH    s0,0($0)      > SH    s0,2(fp)
  %h704201A0: > JAL   LM^FREELIST   LI    a0,54         @ LH    t5,2(fp)
  %h704201AC:   ADDI  t0,fp,2       SRL   s1,t0,1         ADDIU sp,sp,6
  %h704201B8:   SH    t5,-4(sp)
```

5. This command lists the TNS/R instructions from an Accelerated program for the statement at #80.

```
-PTALIN-ICODE AT #80 FOR 1 S

#80

%h70420230:   LI    s0,1      LI    s2,2      LWL   s3,6(fp)
%h7042023C:   LWR   s3,9(fp)  ADDIU sp,sp,10  SH    s0,-8(sp)
%h70420248:   SH    $0,-6(sp) SH    s2,-4(sp) SWL   s3,-2(sp)
%h70420254:   SWR   s3,1(sp)  JAL  PROC1      LI    a0,69
```

6. This command lists the correspondence between TNS instructions and TNS/R instructions starting at TNS address %75 and extending for 11 TNS instructions.

```
-PTALIN-ICODE %75 FOR 11 BOTH

%000100:   > LDI    +000       %h70420230: LI    s0,1
%000101:     LDI    +002       %h70420234: LI    s2,2
%000102:     LDD   L+003       %h70420238: LWL   s3,6(fp)
%000103:     PUSH    744       %h7042023C: LWR   s3,9(fp)
%000104:     PCAL    PROC1     %h70420040: ADDIU sp,sp,10
                               %h70420044: SH    s0,-8(sp)
                               %h70420248: SH    $0,-6(sp)
                               %h7042024C: SH    s2,-4(sp)
                               %h70420250: SWL   s3,-2(sp)
                               %h70420254: SWR   s3,1(sp)
                               %h70420258: JAL  PROC1
%000105:  @ LDI    +001        %h70420260: LI    s01
%000106:    LDI    +000        %h70420264: LI    s2,2
%000107:    LDI    +002        %h70420268: LWL   s3,6(fp)
%000110:    LDD   L+003        %h7042026C: LWR   s3,9(fp)
%000111:    PUSH    744        %h70420270: ADDIU sp,sp,10
%000112:    PCAL    PROC2      %h70420274: SH    s0,-8(sp)
                               %h70420278: SH    $0,-6(sp)
                               %h7042027C: SH    s2,-4(sp)
                               %h70420280: SWL   s3,-2(sp)
                               %h70420284: SWR   s3,1(sp)
                               %h70420288: JAL  PROC2
```

7. This example illustrates the listing of native TNS/R instructions. TNS/R instructions from a previous source line are annotated with a “-” and TNS/R instructions from proceeding lines are annotated with a “+”. Lines containing TNS/R instructions also contain the source file line number that the instruction is for.

```
-PROGRAM-ICODE AT #PROC FOR 3 STATEMENTS

 #10
    10.000    %h700002B0:  addiu   $sp,$sp, -32
    10.000    %h700002B4:  sw      $4,32($sp)
    10.000    %h700002B8:  sw      $5,36($sp)
 #17
    17.000    %h700002BC:  lw      $15,36($sp)
    17.000    %h700002C0:  lw      $14,32($sp)
  - 10.000    %h700002B4:  sw      $6,40($sp)
    17.000    %h700002C8:  lw      $25,40($sp)
  - 10.000    %h700002CC:  sw      $7,44($sp)
    17.000    %h700002D0:  lw      $9,44($sp)
    17.000    %h700002D4:  add     $24,$14,$15
  + 18.000    %h700002D8:  sw      $11,$48($sp)
    17.000    %h700002DC:  add     $8,$24,$25
  - 10.000    %h700002E0:  sw      $31,28$sp)
    17.000    %h700002E4:  add     $10,$8,$9
    17.000    %h700002E8:  lw      $10,16($gp)
 #18
    18.000    %h700002EC:  move    $5,$15
    18.000    %h700002F0:  move    $4,$14
    18.000    %h700002F4:  move    $6,$15
    18.000    %h700002F8:  move    $7,$9
    18.000    %h700002FC:  jal     0x70000290
    18.000    %h70000300:  sw      $11,$16($sp)
-PROGRAM-
```

IDENTIFIER

The IDENTIFIER command displays information about the internal characteristics of a given data location or of all data locations in one or more scope units. The IDENTIFIER command is a synonym for the INFO IDENTIFIER command.

```
IDENTIFIER  { * |  identifier-spec  }


identifier-spec:   one of

    [  scope-path.  ]  identifier  [  .identifier  ]...
    scope-path
    #data-block
    ##GLOBAL
```

`*`

specifies all identifiers in the scope unit identified by the current scope path.

identifier-spec

specifies a given identifier or group of identifiers.

`[ scope-path. ]  identifier [ .identifier ]`

specifies a unique identifier.

scope-path

specifies all identifiers in a given scope unit.

#data-block

specifies all identifiers in a given data block.

`##GLOBAL`

specifies all identifiers in the global data block implicitly named by TAL.

Related Commands

- INFO IDENTIFIER on page 6-105
- MATCH with the IDENTIFIER option on page 6-144

IF

The IF command provides conditional execution of an Inspect command.

```
IF expression THEN { command | alias-name }
```

expression

specifies the expression that must evaluate to TRUE (that is, a nonzero value) before Inspect will execute the command or alias following THEN.

command

is any Inspect command except FA, FB, FC, FK, or XC.

alias-name

is the name of a previously defined alias. The replacement string for the given alias must be a valid command list.

Usage Considerations

- The IF Command and the BREAK THEN Clause

  You can use the IF command within the THEN clause of the BREAK command to execute a break action conditionally. Given the following breakpoint, Inspect will modify A only if its value is >100 when the breakpoint is triggered:

  `BREAK` *`tax-section`* `THEN "D a;IF a>100 THEN M a=1;R"`

  After the break event, Inspect displays the value of A, resets it to 1 if it exceeds 100, and then resumes execution. If A is less than 100, execution will not be resumed.

  Note the use of quotes when IF is used as part of the BREAK command compared to when it is used IN an IF.

```
IF J=3 THEN RESUME
IF J=3 THEN DISPLAY 'Not far enough';RESUME
```

Note that using the THEN command in the THEN clause of a BREAK serves a purpose different from the IF clause of the BREAK command. If the condition is specified in the IF clause, Inspect breaks only when the condition is met:

```
BREAK tax-section IF a>100 THEN "D a;M a=1;R"
```

The result is that A is displayed only when its value exceeds 100.You can use the IF command within the BREAK command in the command file.

- If an expression contains a string, each element of the string must be evaluated individually. For example:

Given the TAL declaration:

```
INT Animal[0:3] := "bird";
```

then for an expression to evaluate to TRUE, the IF command must be entered as follows:

```
IF Animal[0] = "b" AND Animal[1] = "i" AND &
   Animal[2] = "r" AND Animal[3] = "d"     &
THEN DISPLAY Tree;RESUME
```

INFO

The INFO command displays information about various types of items in the current program.

This diagram shows the complete syntax for the INFO command and its clauses. Detailed descriptions of the clauses, including usage considerations and examples, are presented in the following subsections.

```
INFO info-item


info-item:   one of

   IDENTIFIER { * | identifier-spec }
   LOCATION [ code-location | * [ SCOPE { scope-spec } ] ]
   OBJECTFILE [ FILE filename ]
   OPEN[S] [ { * | file-list } [ DETAIL ] [ file-type ] ]
   SAVEFILE [ FILE filename  ]
   SCOPE [ scope-spec ]
   SEGMENT[S] [ * | segment-id ]
   SIGNAL[S] [ * | signal-id [, signal-id...] ]


identifier-spec:   one of

   scope-path
   [ scope-path. ] identifier [ .identifier ]...
   #data-block
   ##GLOBAL


scope-spec:  one of

   scope-path
   scope-ordinal


   #file-list:

   file-number [, file-number ]...
```

INFO IDENTIFIER

The INFO IDENTIFIER command displays information about the internal characteristics of a given data location or of all data locations in one or more scope units, including native symbols.

```
INFO IDENTIFIER { * | identifier-spec }


identifier-spec:   one of

   [ scope-path. ] identifier [ .identifier ]...
   scope-path
   #data-block
   ##GLOBAL
```

`*`

specifies all identifiers in the scope unit identified by the current scope path.

`identifier-spec`

specifies a given identifier or group of identifiers.

`[ `*`scope-path`*`. ] `*`identifier`*` [ .`*`identifier`*` ]...`

specifies a unique identifier.

`scope-path`

specifies all identifiers in a given scope unit.

`#data-block`

specifies all identifiers in a given data block.

`##GLOBAL`

specifies all identifiers in the global data block implicitly named by TAL.

Usage Considerations

- When You Can Use INFO IDENTIFIER

  The INFO IDENTIFIER command can be used for active and for inactive scope units in processes, PATHWAY servers, and save files. In a PATHWAY requester program, the INFO IDENTIFIER command can be used only for active scope units.

- Types of Entities

  The form of the output produced by INFO IDENTIFIER for a given identifier depends on the type of entity the identifier denotes. The distinct types of entities that INFO IDENTIFIER recognizes are:

BLOCK DATA	BLOCK NAME	CONDITION
DEFINED TYPE	ENTRY	FILE NAME
FORMAT	IN CONTAINING SCOPE	INDEX
INLINE PROC	LABEL	LITERAL
MACRO	MNEMONIC NAME	NAMED CONST
PROC	PROCEDURE PARAM	REGISTER
SCREEN	SUBPROC	VARIABLE

- INFO IDENTIFIER Presentation for Variables

  A typical program contains more identifiers for variables than for any other entity class. Here is the form of the INFO IDENTIFIER report for a variable:

```
identifier: VARIABLE
storage^info:
TYPE=data-type, ELEMENT LEN=len BITS, UNIT SIZE=size ELEMENTS, SCALE=scale
access^info:
location
dimension^info:
dimension
structure^info:
PARENT=parent, CHILD=child, SIBLING=sibling
```

The information that INFO IDENTIFIER provides is:

`storage^info`	Reports the data type, the size of the element in bits, the number of elements that make up the entity, and any scale factor.
`access^info`	Reports the location of the entity. If the entity is on a word-aligned boundary, INFO IDENTIFIER displays the location in words; otherwise, INFO IDENTIFIER displays the location in words and bytes.
`dimension^info`	Reports the dimensions of the entity, if it is an array.
`structure^info`	Identifies the parent (group item to which the entity belongs), the child (first entity contained within the entity whose attributes are being displayed), and the sibling (next entity in the same group as that whose attributes are being displayed).

The data types that appear in `storage^info` is:

`BIN SIGN`	Binary signed
`BIN UNSIGN`	Binary unsigned
`BYTE STRUCT`	Byte-addressed structure
`CHAR`	Character

`COMPLEX`	Complex
`DEFINED TYPE`	Defined type
`LOGICAL`	Logical
`NUM LD EM`	Numeric, sign leading, embedded
`NUM LD SP`	Numeric, sign leading, separate
`NUM TR EM`	Numeric, sign trailing, embedded
`NUM TR SP`	Numeric, sign trailing, separate
`NUM UNSIGN`	Numeric unsigned
`PAK SIGNED`	Packed numeric signed
`PAK UNSIGN`	Pack numeric unsigned
`REAL`	Real
`WORD STRUCT`	Word-addressed structure

Note that these data types do not correspond exactly to any one language's types. Because Inspect supports several different programming languages, it uses generic terms to denote data types.

Other entity classes have other forms, depending on the types of attributes each entity class has.

If a portion of the report does not apply to a particular variable, that portion is not displayed. For example, if a variable has no parent, the PARENT entry is omitted from the report.

Related Commands

- DISPLAY on page 6-33
- IDENTIFIER on page 6-102
- MATCH with the IDENTIFIER option on page 6-144

Examples

1. A COBOL or SCREEN COBOL alphanumeric data item named THICKNESS can, for example, have these attributes:

```
-PRG-INFO IDENTIFIER thickness
THICKNESS: VARIABLE
storage^info:
TYPE=CHAR, ELEMENT LEN=8 BITS, UNIT SIZE=24 ELEMENTS
access^info:
'L' + %22S WORDS
dimension^info:
[1:2]
structure^info:
PARENT=LUMBER-TABLE,CHILD=WIDTH
```

2. A FORTRAN real variable named P will have these attributes:

```
-PRG-INFO IDENTIFIER p
P: VARIABLE
storage^info:
TYPE=REAL, ELEMENT LEN=32 BITS, UNIT SIZE=1 ELEMENTS
access^info:
'L' + %15  WORDS
```

3. A TAL integer array named D^ARRAY could have these attributes:

```
-PRG-INFO IDENTIFIER d^array
D^ARRAY: VARIABLE
storage^info:
TYPE=BIN SIGN, ELEMENT LEN=16 BITS, UNIT SIZE=1 ELEMENTS
access^info:
'L' + 1  WORD
dimension^info:
[0:9]
```

4. A COBOL numeric unsigned data item named PRIMARY-CPU could have these attributes:

```
-PRG-INFO IDENTIFIER primary-cpu
PRIMARY-CPU: VARIABLE
storage^info:
TYPE=NUM UNSIGNED, ELEMENT LEN=8 BITS, UNIT SIZE=1 ELEMENTS
access^info:
'L' + %20 WORDS + 1 BYTE
```

In this example, PRIMARY-CPU is not on a word boundary; therefore, Inspect displays access^info showing the specific byte.

INFO LOCATION

The INFO LOCATION command displays information about a statement in the current program. Additional information is listed for accelerated programs, including the effects that accelerator optimizations have on source statements.

```
INFO LOCATION [ code-location | * [ SCOPE { scope-spec } ] ]


scope-spec:  one of

   scope-path
   scope-ordinal
```

code-location

specifies the statement you want information about.

*

specifies all statements in the current scope unit.

scope-spec: one of

scope-path

specifies a scope unit by name.

scope-ordinal

specifies a scope unit by its scope number as displayed by the TRACE command.

Default Values

If you enter INFO LOCATION alone, Inspect uses the current scope path to determine what statement to use:

- If the current scope path denotes an active scope unit, the INFO LOCATION command provides information about the statement containing the current code location in the scope unit.
- If the current scope path denotes an inactive scope unit, the INFO LOCATION command provides information about the first statement in the scope unit.

Usage Considerations

- Location Information

  INFO LOCATION displays these information:

  - The name of the scope unit containing the statement.
  - The compilation name and modification timestamp of the source file from which the statement was compiled. If a SOURCE ASSIGN has been applied to the file, the name of the ASSIGN file is also listed.

    This illustrates the form of this output:

```
  Scope: scope-name

   Compile File:   filename             Modified: timestamp
 [ ASSIGN File:    filename                                ]
 [ Source System:  system                                  ]
```

  - The statement number of the statement.
  - The EDIT line number at which the statement starts.
  - The offset in words of the statement from the base of its containing scope unit.
  - An optimize verb denoting what optimization (if any) the compiler performed on the statement. Possible optimize verbs are Deleted and Merged. Deleted indicates that the compiler deleted the statement or incorporated it in another

statement. Merged indicates that the compiler merged the statement with one or more other statements.

You cannot set a breakpoint on a deleted statement; you can set one on a merged statement. However, when a breakpoint on a merged statement generates a break event, Inspect displays this message:

```
** Inspect warning 198 ** Results might be unexpected due to optimization
```

Usage Considerations for Accelerated Programs

- For accelerated programs, output contains an additional column titled “Register-Exact.” “Yes” is listed in this column if TNS registers are valid at the beginning of the statement.
- Register-exact points do not exist at the beginning of many statements For programs translated at the ProcDebug level of optimization.
- The INFO LOCATION command does not report register-exact points that exist within a statement.
- If the location of a statement is not a memory-exact point, “Deleted” is listed in the Optimize column.

Note. “deleted” does not mean that the actions of your statements have been deleted, but that optimization has merged the statement with another, which resulted in the statement not being available for debugging.

Related Commands

- ADD SOURCE ASSIGN on page 6-14
- SELECT SOURCE SYSTEM on page 6-169
- SOURCE ASSIGN on page 6-202
- SOURCE SYSTEM on page 6-211

Example

This example output illustrates, for accelerated programs, the inclusion of the indication of whether or not a statement is a register-exact point.

```
Scope: M

Compile File:  $GOLF.GREEN.TALIN  Modified: 1988-10-03
12:15:20.02

                Word
Num    Line    Offset   Optimize   Verb   Register-exact
1      #41     %0
2      #52     %1       Deleted
3      #52     %3
4      #52     %6                         Yes
5      #64     %10
```

INFO OBJECTFILE

The INFO OBJECTFILE command displays information about the current program's object files, or any specified object file. It provides additional information for object files created by the accelerator and native object files.

Note. The INFO OBJECTFILE command is invalid for PATHWAY requester programs.

```
INFO OBJECTFILE [  FILE filename  ]
```

filename

is the name of an object file. Depending on the current systype of Inspect, filename may either be a Guardian filename or an OSS pathname.

Usage Considerations

- Inspect need not be debugging a program to use the FILE clause form of this command.
- INFO OBJECTFILE does not read TNS/E native object files. An error message is displayed if you enter INFO OBJECTFILE and specify a TNS/E object file. Instead, you can use TNSVU to examine safepoints and corresponding TNS and TNS/E instructions.

Related Commands

- INFO SAVEFILE on page 6-120
- LIST PROGRAM on page 6-137

Output

This output template illustrates the INFO OBJECTFILE command:

```
[ Library | Program]Object File: filename

                General Information

[             BINDER Region:  YES | NO                         ]*1
[          BINDER Timestamp:  timestamp                        ]*1
[             NLD Timestamp:  timestamp                        ]*3
[                Data Pages:  integer                          ]*1
                Debugger:  DEBUG | INSPECT
          INSPECT Region:  YES | NO
             System Type:  GUARDIAN | OSS
         Process Subtype:  integer
     Program File Segment:  integer WORDS
          Highrequesters:  ON | OFF
                Runnamed:  ON | OFF
                 Highpin:  ON | OFF
               Saveabend:  ON | OFF
                   SRL Name:  name                             ]*4
                  SRLs Used:  name                             ]*4
                   Segments:  integer                          ]*1
                     Target:  target                           ]*1

[             Accelerator Information                          ]*2
[                                                              ]*2
[     Accelerated Execution:  ENABLED | DISABLED               ]*2
[              Optimization:  PROCDEBUG | STMTDEBUG | UNKNOWN]*2
[            Global Options:  accelerator options              ]*2
[               Timestamp:  timestamp                          ]*2
[                 Version:  timestamp                          ]*2
```

timestamp is of the form:

```
YYYY-MM-DD HH:MM:SS.DD
```

accelerator options: one from each line of:

ATOMIC_ON	ATOMIC_OFF
OVTRAP_ON	OVTRAP_OFF
INHERITSCC_ON	INHERITSCC_OFF
SAFEALIASINGRULES_ON	SAFEALIASINGRULES_OFF
TRUNCATEINDEXING_ON	TRUNCATEINDEXING_OFF

target is one of:

```
TNS
TNS/R
ANY
UNSPECIFIED
UNKNOWN
```

- Accelerator-related information is listed only for object files that have been processed by an accelerator (Axcel on TNS/R systems, OCA on TNS/E systems).

- If a program has a user library, information is listed for the program file and the library file.
- If the filename parameter is omitted, Inspect will display information about all object files used by the current process (user code plus all libraries plus all system files). Items labeled with “*1” apply only to TNS object files, items with “*2” apply only to accelerated object files, items with “*3” apply only to native object files, and items with “*4” apply only to native SRL object files.

Examples

1. This example illustrates the information the INFO OBJECTFILE command presents for a program accelerated with the ProcDebug option.

```
-PROGRAM-INFO OBJECTFILE
Program Object File: \SYSTEM.$DISK.SUBVOL.OBJECT

                   General Information

               BINDER Region: YES
            BINDER Timestamp: 1992-08-13 17:46:40.57
                  Data Pages: 64
                    Debugger: INSPECT
              INSPECT Region: YES
                System Type:  Guardian
             Process Subtype: 0
        Program File Segment: 0 WORDS
              Highrequesters: OFF
                    Runnamed: OFF
                     Highpin: OFF
                   Saveabend: OFF
                    Segments: 1
                      Target: UNSPECIFIED

                 Accelerator Information

       Accelerated Execution: ENABLED
                Optimization: PROCDEBUG
              Global Options: ATOMIC_OFF, INHERITSCC_OFF, OVTRAP_ON,
                              SAFEALIASINGRULES_OFF, TRUNCATEINDEXING_ON
                   Timestamp: 1992-08-13 18:29:17.52
                     Version: 1992-02-25 10:18:32.46
```

2. This example illustrates the information the INFO OBJECTFILE command presents for a TNS/R program.

```
-DEMO-INFO OBJECT
Program Object File: \FOLK.$LORE.EXAMPLE.DEMO

                   General Information

               NLD Timestamp: 1995-08-11 14:27:01.00
                    Debugger: DEBUG
             Process Subtype: 0
        Program File Segment: 0 WORDS
              Highrequesters: OFF
                    Runnamed: OFF
                     Highpin: OFF
                   Saveabend: OFF
```

3. This example illustrates the information the INFO OBJECTFILE command presents for a TNS/R program using SRLs.

```
-DEMO-INFO OBJECT
Program Object File: \FOLK.$LORE.EXAMPLE.DEMO

                   General Information

                NLD Timestamp: 1995-08-11 14:27:01.00
                     Debugger: DEBUG
              Process Subtype: 0
         Program File Segment: 0 WORDS
               Highrequesters: OFF
                     Runnamed: OFF
                      Highpin: OFF
                    Saveabend: OFF
                    SRLs Used: ZCRTLSRL
                              ZCRESRL

 Library Object File: \FOLK.$SYSTEM.SYS01.ZCRESRL

                   General Information

                NLD Timestamp: 1995-10-16 10:33:41.00
                     Debugger: DEBUG
              Process Subtype: 0
         Program File Segment: 0 WORDS
               Highrequesters: OFF
                      Highpin: ON
                    Saveabend: OFF
                     SRL Name: ZCRESRL

 Library Object File: \FOLK.$SYSTEM.SYS01.ZCRTLSRL

                    General Information

              NLD Timestamp: 1995-10-19 13:39:13.00
                   Debugger: DEBUG
            Process Subtype: 0
       Program File Segment: 0 WORDS
             Highrequesters: OFF
                   Runnamed: OFF
                    Highpin: ON
                  Saveabend: OFF
                   SRL Name: ZCRTLSRL
                  SRLs Used: ZCRESRL
                             ZI18NSRL
                             ZOSSKSRL
                               ZICNVSRL
```

INFO OPENS

The INFO OPENS command shows the status of files that have been opened by the current program. Although all opened files contain a Guardian file number, you can refer to opened OSS files using OSS concepts and terminology.

Note. The INFO OPENS command is invalid for PATHWAY requester programs.

```
INFO OPENS  [ { * | file-list } [ DETAIL ] [ file-type ] ]

file-list

   file-number  [, file-number  ]...

file-type:  one of

   FORTRAN   FD   GUARDIAN
```

`*`

requests the status of all files opened by the current program.

`file-list`

requests the status of specific files. *`file-list`* is a list of file numbers identifying the desired files.

`file-number`

specifies a single file. *`file-number`* can be:

A COBOL FD name
An expression that evaluates to an integer value
A data location identifying an integer value

Inspect interprets the integer values as file-system file numbers, unless you specify the F clause; Inspect would then interpret the values as FORTRAN logical-unit numbers. If a FD clause is used, Inspect interprets the values as OSS file descriptors.

`DETAIL`

directs Inspect to display the maximum available information for the specified files.

`file-type`

indicates to Inspect what type of file to display. *`file-type`* can be one of the following:

`FORTRAN`

indicates that *`file-number`* values specify `FORTRAN` logical-unit numbers, not file system file numbers. Files without `FORTRAN` logical units will not be displayed.

`FD`

indicates that *`file-number`* values specify OSS file descriptors. Files without OSS file descriptors will not be displayed.

`GUARDIAN`

indicates that *`file-number`* values specify Guardian system file numbers. Files without Guardian file system numbers will not be displayed.

Default Values

- The default file type is Guardian file system numbers.
- If you enter the INFO OPENS command without parameters, Inspect displays the Guardian file number, physical file name, and last error of all files opened by the current program.

Usage Considerations

- Open File Information

  INFO OPENS displays these information:

  - The file-system file number (as returned by the OPEN system procedure)
  - The physical file name
  - The file-system error number associated with the last operation on the file

- The fields of the INFO OPENS command DETAIL clause

  This table describes information Inspect displays for the INFO OPENS command, DETAIL clause.

DETAIL Field Name	**Field Description** (page 1 of 2)
FILE NUMBER	File number of the opened file.
LAST ERROR	Error of last operation on file.
NAME	Name of file.
PRE-D00 MSGS	($RECEIVE only) reading C-series format messages or D-series format.
DEVICE TYPE	Device type of the device associated with the file.
OPEN FLAGS	Flags used to open the file with.
REQUESTS OUT	Type of the oldest outstanding nowaited operation.
LOG DEV	Logical device number of the device where the file resides.
FILE CODE	File code of file.
CUR REC	Current record pointer.
PEXT SIZE	Primary extent size.
NEXT REC	Next record pointer.

DETAIL Field Name	**Field Description** (page 2 of 2)
SEXT SIZE	Secondary extent size.
EOF	Relative byte address of the end-of-file location.
LAST MODIFIED	Time the file was last modified.

For more information, see the description of FILEINFO in *Guardian Procedure Calls Reference Manual*.

- Using the DETAIL clause

  When you use the DETAIL clause, Inspect displays system message information indicating whether $RECEIVE is reading C-series format messages or D-series format messages.

- Most Recent File Accessed

  If you enter the INFO OPENS command with a *file-number* of -1, Inspect reports the file-system error number for the last CREATE, PURGE, or OPEN.

- Shared Files

  Because the Common Run-Time Environment (CRE) provides multiple connections to a single open of a file, the INFO OPENS command shows only one open for each standard file, regardless of how many connections the CRE had granted for the file. If none of the routines have the standard file open, the file does not appear in the file list displayed by Inspect.

- Unnamed File Types

  OSS pipes and temporary files (files created using *tmpfile*()) do not have a name. Inspect will display the text “<<unknown >“ as the name for these types of files.

- OSS pathnames

  Inspect will display the OSS pathname for all OSS opens that have a name, even if a Guardian file system number was specified.

Output

The DETAIL clause of the INFO OPENS command produces output of this form for OSS file descriptors.

```
Descriptor: integer
Number: integer
Name:   pathname

                  OSS File Information

                        File Type:  file-type
                       Last Error:  integer
                    Close on Exec:  ON | OFF

                 OSS Disk File Information

                    Serial Number:  double
                        Device ID:  double
                             RDev: double
                       Link Count: integer
                              UID: uid
                              GID: gid
                      End of file: double
                 Access Timestamp: timestamp
                 Change Timestamp: timestamp
```

Note. The detailed output for OSS files will be slightly different depending on whether Guardian file numbers or OSS file descriptors are specified. If Guardian file numbers are specified, then the list of OSS file descriptors that correspond to this file will be displayed. If OSS file descriptors are specified, then the Guardian file number that corresponds to this file will be displayed. In addition, lines bracketed with [] might not be listed for some file types (such as non-disk files) or when examining pre-D20 save files.

Examples

1. This example lists the information for each open Guardian file. By default, the INFO OPENS command lists information for all open Guardian files.

```
-I001OBF0-INFO OPEN *
File                                              Last
Number   Filename                                 Error
    1    \CUBS.$BOB.QAT9252I.NEW                     0
    2    \CUBS.$BOB.QAT9252I.NEW                     0
    3    \CUBS.$BOB.QAT9252I.I000DAT0                0
```

2. For FORTRAN programs where logical units have been defined, the F clause lists information for all defined logical units. Logical units followed by “Not Assigned” have not been assigned a Guardian file number.

```
-I001OBF0-INFO OPEN * f
Logical   File                                Last
Unit      Number   Filename                   Error
   0               Not Assigned
   1         1     \CUBS.$BOB.QAT9252I.NEW        0
   2         2     \CUBS.$BOB.QAT9252I.NEW        0
   3               Not Assigned
   4               \CUBS.$ZTNT.#PTY23
   5               \CUBS.$ZTNT.#PTY23
   6               \CUBS.$ZTNT.#PTY23
   7               Not Assigned
   8               Not Assigned
   9               \CUBS.$BOB
  10         3     \CUBS.$BOB.QAT9252I.I000DAT0      0
```

3. This example shows INFO OPENS output for a disk file. Information is organized into two sections with general information listed first followed by information specific to that disk file. The true value of items listed mnemonically is listed parenthetically. Also note that the current record pointer indicates where you are currently in the file.

```
-I001OBF0-INFO OPEN 1, dNumber: 1Name:   \CUBS.$BOB.QAT9252I.NEW
General File Information                  Device Type: 3
Device Subtype: 18               File Type: UNSTRUCTURED (0)
Last Error: 0          Logical Device Number: 35                 Open
Access: READ-WRITE (0)                Open Exclusion: SHARED (0)
Open NOWAIT Depth: 0                   Open Options: %0             Open
Sync Depth: 0           Outstanding Requests: 0        Physical Record
Length: 4096 Bytes                  Disk File Information
Block Length: 0 Bytes                   End of file: 264 Bytes
Current Record Pointer: 132          Extent Size: 2 Pages, 2 Pages
Flags: AUDITED, DEMOUNTABLE, WRITE-THRU                  File Code: 0
Logical Record Length: 0 Bytes               Maximum Extents: 0
Modification Timestamp: 1993-06-03 11:19:18.870.999           Next Record
Pointer: 264                      Partitions: 0
```

4. This example shows Inspect’s display for the INFO OPENS command without the DETAIL, F, or FD clause. Note that Guardian file numbers 1,2, and 3 have an OSS pathname, instead of their equivalent Guardian file names.

```
-PROGRAM-INFO OPENS *
     #1      /dev/tty0                           #0000
     #2      /dev/tty1                           #0000
     #3      /usr/src/file.c                     #0000
     #4      \CUBS.$ERROR.SUBVOL.FILE1           #0000
```

5. Using the previous example, the following output illustrates what the INFO OPENS command displays when the FD clause is used. Only files that have an OSS file descriptor associated with them are displayed.

```
-test4-INFO OPENS * FD
File        File                               Last
 Descriptor Number   Filename                   Error
   0           4    /G/ztnt/#pty0025               0
   1           5    /G/ztnt/#pty0025               0
   2           6    /G/ztnt/#pty0025               0
   3           1    /usr/morris/test1.c            0
```

INFO SAVEFILE

The INFO SAVEFILE command displays information about the save file for the current program (if any) or a specified save file.

```
INFO SAVEFILE  [ FILE  filename ]
```

filename

is the name of a save file. Depending on the current systype of Inspect, filename may be either a Guardian filename or an OSS pathname.

Usage Considerations

- When a process with its SAVEABEND attribute set terminates abnormally, DMON automatically creates a save file in the volume and subvolume containing the program file. The save file that DMON creates has a file code of 130 and a name of the form ZZSA*dddd*, where *dddd* is a number chosen by DMON.
- This command is useful for determining why a save file was created and the program and library file information associated with a save file.
- Inspect need not be debugging a program to use the FILE clause form of this command.
- INFO SAVEFILE cannot read TNS/E native snapshot files. If you enter an INFO SAVEFILE command and specify a TNS/E native snapshot, an error message is displayed: “Inspect cannot read TNS/E snapshot files.”

Related Commands

- INFO OBJECTFILE on page 6-111
- LIST PROGRAM on page 6-137
- SAVE on page 6-160

Output

The INFO SAVEFILE command produces output of this form:

```
Save File: savefile name

[                            Cause: COMMAND | ABEND | SIGNAL ]
                           Creator: CRUNCH | DMON | INSPECT
                Creation Timestamp: timestamp
                 Creator's User ID: userid
                  Guardian Version: Lnn
[                      Trap Number: integer                  ]  *1
[                    Signal Number: integer                  ]  *1
[                      Wait Status: integer                  ]  *2
                         Processor: family
                      Program File: filename
          Program Binder Timestamp: timestamp
    Program Modification Timestamp: timestamp
                            System: name (number)
[               Truncated File Info: YES                      ]
[            Truncated Segment Info: YES                      ]

[                        Library Information                  ]
[                      Library File: filename                 ]
[          Library Binder Timestamp: timestamp                ]
[    Library Modification Timestamp: timestamp                ]
```

family is one of:

```
TNS
TNS/R
```

processor is one of:

```
TNSII
TXP
VLX
CLX
Cyclone
NSR-L
NSR-N
```

- Fields listed in square brackets are only listed when appropriate.
- Information about the cause of save file creation is only listed for C30 or later versions of save files.
- Truncated file information is listed when it was not possible to save information about all open files in the save file.
- Truncated segment information is listed when it was not possible to save information about all data segments in the save file.
- The Processor field is only listed for versions of save files in which this information is stored (C30 or later).
- Library information will be repeated for each library used by the process. Items marked with “*1” are mutually exclusive (only one will be displayed) and will only

be displayed if the process terminated due to a trap (or signal if a native process), and items marked with “*2” are for OSS processes only.

Example

An example of information the INFO SAVEFILE command produces is:

```
-DEMO-INFO SAVEFILE
Save File: \FOLK.$LORE.HELP.DEMOS
                            Cause: COMMAND
                          Creator: INSPECT
               Creation Timestamp: 1995-11-12 22:46:58.269.416
                Creator's User ID: SUPER.SUPER (255,255)
               Guardian 90 Version: D40
                        Processor: TNS/R (NSR-L)
                     Program File: \FOLK.$LORE.EXAMPLE.DEMO
            Program Link Timestamp: 1995-08-11 14:27:01.000.000
    Program Modification Timestamp: 1995-08-11 14:25:41.048.850
                           System: \FOLK (241)

                        Library InformationFile:

 \FOLK.$SYSTEM.SYS01.ZCRESRL
                  Link Timestamp: 1995-10-16 10:33:41.000.000
          Modification Timestamp: 1995-10-16 10:33:49.530.000

 File: \FOLK.$SYSTEM.SYS01.ZCRTLSRL
                 Link Timestamp: 1995-10-19 13:39:13.000.000
         Modification Timestamp: 1995-10-23 22:43:11.030.000

File: \FOLK.$SYSTEM.SYS01.ZOSSKSRL
                 Link Timestamp: 1995-10-19 11:20:36.000.000
        Modification Timestamp: 1995-10-19 11:20:40.150.000

 File: \FOLK.$SYSTEM.SYS01.ZI18NSRL
                Link Timestamp: 1995-10-16 21:01:17.000.000
        Modification Timestamp: 1995-10-18 08:22:55.850.000

 File: \FOLK.$SYSTEM.SYS01.ZICNVSRL
                Link Timestamp: 1995-10-16 10:06:42.000.000
        Modification Timestamp: 1995-10-18 08:15:38.150.000
```

INFO SCOPE

The INFO SCOPE command displays information about a given scope unit in the current program, including the optimization level a TNR/S native program was compiled with.

Note. The INFO SCOP command is invalid for PATHWAY requester programs.

```
INFO SCOPE [ scope-spec ]


scope-spec:   one of

   scope-number
   scope-path
   #data-block
   ##GLOBAL
```

scope-spec

specifies the scope unit you want.

scope-number

specifies a scope unit by its scope number as displayed by the TRACE command.

scope-path

specifies a scope unit by name. INFO SCOPE displays information about the last scope unit specified in the scope path.

#data-block

specifies a named data block as the scope unit.

##GLOBAL

specifies the implicitly named global data block in TAL as the scope unit.

Default Value

If you enter INFO SCOPE alone, Inspect displays information about the scope unit denoted by the current scope path.

Related Commands

- ADD PROGRAM on page 6-10
- ADD SOURCE ASSIGN on page 6-14
- LIST PROGRAM on page 6-137
- MATCH with the SCOPE option on page 6-144
- SET RADIX on page 6-181
- TRACE on page 6-219

Output

The INFO SCOPE command produces output of this form:

```
Scope Name: name
Source File: file
Modification timestamp: timestamp
Compilation timestamp: timestamp
Language: lang
Type: CODE
Base: integer
Entry: integer
Length: integer words
[Optimization level: integer]        *1
```

Items labelled with “*1” are only present for native code scopes.

Examples

1. This example shows what information the INFO SCOPE command presents for a code block:

```
-PROGRAM-INFO SCOPE #MAIN1A
Scope Name: MAIN1A
Source file: \SYS.$DATA1.TALSUBV.A001TAL0
Modification timestamp: 1987-02-27 18:05:28.87
Compilation timestamp: 1992-10-23 17:12:26.580.000
Language: TAL
Type: CODE
Base: 2744
Entry: 3277
Length: 2594 words
Attributes: MAIN
```

   Note that base, entry, and length are shown in the current output radix.

2. This example shows what information the INFO SCOPE command presents for a data block:

```
-PROGRAM-INFO SCOPE #m0
Scope Name: M0
Compilation timestamp: 1992-10-23 17:12:26.580.000
Language: TAL
Type: DATA
Location: G+90
Size: 30 bytes
```

3. This example illustrates the information presented by the INFO SCOPE command for a TNS/R native program at optimization level zero.

```
-DEMO-INFO SCOPE
Scope Name: main
Source file: \FOLK.$LORE.EXAMPLE.DEMO
Compilation timestamp: 1995-08-11 14:19:18.000.000
Language: C
Type: CODE
Base: 939524552
Entry: 939524552
Length: 40 words
Optimization Level: 0
```

4. This example illustrates the information presented by the INFO SCOPE command for a TNS/R native program at optimization level two.

```
-OPT2SCOP-INFO SCOPE
Scope Name: OPT2SCOP
Source file: \FOLK.$LORE.EXAMPLE.OPT2SCOP
Modification timestamp: 1995-11-12 23:18:31.00
Compilation timestamp: 1995-11-12 23:19:09.000.000
Language: TAL
Type: CODE
Base: 939524552
Entry: 939524552
Length: 114 words
Attributes: MAIN
Optimization Level: 2
** Inspect warning 397 ** Optimization level for scope OPT2 is not supported
by Inspect
```

INFO SEGMENTS

The INFO SEGMENTS command displays information about the extended segments allocated for or by the current program.

```
INFO SEGMENT[S] [ * | segment-id ]  [[, ] DETAIL ]


segment-id:   one of

   integer
   data-location
```

*

requests information for all segments the current program has allocated.

segment-id

is a 16-bit integer.

Default Value

If you enter INFO SEGMENTS alone, Inspect displays information on segments the process has allocated.

Usage Considerations

- Segment Information

  When Inspect displays segment information, it uses this format:

```
Seg ID   Length         Description          Type              Swap File
  nnnn nnnnnnnn  xxxxxxxxxxxxxxxxx  xxxxxxxxxxxxxxxxx  xxxxxxxxxxxxxxxxxxxxxx
```

  The description field describes the type of the extended segment: user defined, language defined, or one of the reserved system segments.

  The type field indicates the type of the segment (CODE or DATA) and any special attributes of the segment. Possible attributes are:

  EXTENSIBLE
  READONLY
  RESIDENT
  SHARED *num-users*

  If all attributes will not fit in the field on one line, some are printed on the following line.

- Warnings and Errors

  If no segments are allocated, Inspect displays the warning:

```
** Inspect warning 239 ** No segments are allocated
```

  If a specified segment is not allocated, Inspect displays the warning:

```
** Inspect warning 40 ** Segment not allocated
```

Related Command

SELECT SEGMENT

Output

This output template illustrates the information the INFO SEGMENT command displays without the DETAIL clause.

```
Seg ID      Length   Description      Type      Swap File
  nnnn       nnnnn  <text>           <type>    <volume>
```

This template shows the information displayed using the DETAIL clause.

```
Segment ID: nnnn (text description)

           Base:  double
 Current Length:  double
    Description:  text
[         Flags:  [ [EXTENSIBLE] [RESIDENT] [READONLY] [SHARED] ]
          Limit:  double
[     Swap File:  filename ]
           Type:  DATA | CODE
```

Example

This example shows the information that INFO SEGMENTS presents. The "*" annotates the current segment in use by the program and the ">" indicates the current segment selected for viewing by Inspect. If the current user segment and the current Inspect segment are the same, only the "*" annotation will be used.

```
-PROGRAM-INFO SEGMENTS
Seg ID     Length  Description        Type               Swap File
*    1     500000  User defined       CODE READONLY
\SHARK.$TOOLS.TDM.SWAP
                                      SHARED 12
> 1024    1200000  Compiler defined   DATA EXTENSIBLE    \SHARK.$SYSTEM.#4343
```

INFO SIGNALS

The INFO SIGNALS command displays signal information for the current program.

```
INFO SIGNAL[S] [ * | signal-id [, signal-id...] ]
                                       [ [, ] DETAIL ]
```

The INFO SIGNALS command displays signal information for the current program.

```
INFO SIGNAL[S] [ * | signal-id [, signal-id...] ]
                                       [ [, ] DETAIL ]
```

*

requests the status of all signals of the current program.

signal-id

requests the status of a specific signal, one of:

```
SIGABRT         SIGALRM        SIGFPE
SIGHUP          SIGILL         SIGINT
SIGIO           SIGKILL        SIGPIPE
SIGQUIT         SIGRECV        SIGSEGV
SIGTERM         SIGUSR1        SIGUSR2
SIGCHLD         SIGCONT        SIGSTOP
```

```
SIGTSTP         SIGTTIN         SIGTTOU
SIGABEND        SIGLIMIT        SIGSTK
SIGMEMMGR       SIGNOMEM        SIGMEMERR
SIGTIMEOUT
```

Default Value

If you enter INFO SIGNALS alone, Inspect displays information on all the signals of the current program.

Output

This output illustrates the information displayed using the DETAIL clause.

```
    Signal:  signal-id
            Handler:  SIG_DFL | SIG_IGN | SIG_DEBUG | #function-name
               Mask:  NONE | signal-id [, signal-id ...]
              Flags:  flags-word
```

Example

This example shows the information that the INFO SIGNAL command displays without the DETAIL clause. Information for each signal is split across two lines.

```
-ETEST-INFO SIGNAL SIGFPE
Signal
  Handler              Mask                            Flags
SIGFPE(8)
  CRE_TRAP_HANDLER_    (0        0        0        0       ) 0
```

KEY

The KEY command adds a function-key definition or displays one or all function-key definitions in the function-key list for the current Inspect session. The KEY command is synonym for the ADD KEY and LIST KEY commands.

```
KEY[S]  [  key-name  [  [=]  replacement-string  ]  ]


key-name:   one of

   F1   F2    F3    F4    F5    F6    F7    F8
   F9   F10   F11   F12   F13   F14   F15   F16
   SF1  SF2   SF3   SF4   SF5   SF6   SF7   SF8
   SF9  SF10  SF11  SF12  SF13  SF14  SF15  SF16

replacement string:  one of:

   " [ character ]   "
   ' [ character ]   '
```

`key-name`

specifies the function key for which you want to provide or display a definition. Valid function keys include F1 through F16 and shifted F1 (SF1) through shifted F16 (SF16).

`replacement string`

specifies the replacement string to associate with the given function key. The replacement string is a group of zero or more characters enclosed in either quotes (") or apostrophes ('). To include a quote in a quote-delimited replacement string, use a pair of quotes. Likewise, to include an apostrophe in an apostrophe-delimited replacement string, use a pair of apostrophes.

Default Values

- If you do not specify a key-name nor a replacement-string, KEY displays all function-key definitions.
- If you specify a key-name but not a replacement-string, KEY displays the function-key definition for key-name.

Usage Consideration

If you specify key-name and replacement-string, KEY adds a function-key definition for key-name.

Related Commands

- ADD KEY on page 6-9
- DELETE KEY on page 6-30
- FK on page 6-90
- LIST KEY on page 6-136

LIST

The LIST command displays the contents either of the history buffer or of one of the lists that Inspect maintains.

This diagram shows the complete syntax for the LIST command and its clauses. Detailed descriptions of the clauses, including usage considerations and examples, are presented in the following subsections.

```
LIST list-spec


 list-spec:   one of
```

```
   ALIAS[ES] [ alias-name ] [ AS COMMAND[S] ]
   BREAKPOINT[S] [ breakpoint-number | * ] [ options ]
   HISTORY [ command-range ] [ AS COMMAND[S] ]
   KEY[S] [ key-name ] [ AS COMMAND[S] ]
   PROGRAM[S] [ program | * ] [ [ , ] DETAIL ]
   SOURCE ASSIGN[S] [ AS COMMAND[S] ]
   SOURCE OPEN[S]


options:   one of

   [ [ , ]  DETAIL ] | [ AS COMMAND[S] ]


key-name:   one of

   F1    F2    F3    F4    F5    F6    F7    F8
   F9    F10   F11   F12   F13   F14   F15   F16
   SF1   SF2   SF3   SF4   SF5   SF6   SF7   SF8
   SF9   SF10  SF11  SF12  SF13  SF14  SF15  SF16


program:   one of

   program-number
   program-name
```

Usage Consideration

The AS COMMANDS clause directs LIST to display a list as executable Inspect commands; for example:

```
-PRG-LIST ALIASES AS COMMANDS
ADD ALIAS SHOWVARS = "DISPLAY height, width, depth"
ADD ALIAS WATCHVARS = "DISPLAY height, width, depth; RESUME"
```

If you use the AS COMMANDS clause with the OUT clause, you can create a command file. You can then incorporate this file in an OBEY file or the INSPLOCL or INSPCSTM customization files; for example:

```
-PRG-LIST /OUT alias/ ALIASES AS COMMANDS
```

Note that if the file already exists, output will be appended to it rather than overriding existing data.

Related Commands

- ADD on page 6-6
- DELETE on page 6-29
- SELECT on page 6-164

LIST ALIAS

The LIST ALIAS command displays either one or all aliases from the alias list for the current Inspect session.

```
LIST ALIAS[ES] [ alias-name ] [ AS COMMAND[S] ]
```

alias-name

specifies the name of a previously defined alias.

`AS COMMAND[S]`

directs LIST ALIAS to display each alias definition as an ADD ALIAS command.

Default Value

If you do not specify an alias name, LIST ALIAS displays all alias definitions.

Usage Consideration

Aliases are not expanded with the LIST ALIAS command.

Related Commands

- ADD ALIAS on page 6-7
- ALIAS on page 6-17
- DELETE ALIAS on page 6-30

LIST BREAKPOINT

The LIST BREAKPOINT command displays either one or all breakpoints defined in the current program. This command applies to breakpoints set by both Inspect and DEBUG.

```
LIST BREAKPOINT[S] [ breakpoint-number | * ] [ options ]

options:   one of

   [ , ] DETAIL
   AS COMMAND[S]
```

breakpoint-number

specifies the number of a previously defined breakpoint.

*

lists all breakpoints.

`AS COMMAND[S]`

directs LIST BREAKPOINT to display the breakpoint definition as a BREAK command.

Default Values

- If no breakpoint number is specified, LIST BREAKPOINT displays all breakpoints defined in the current program.
- The asterisk (*) lists information for all breakpoints.

Usage Considerations

- Data, low-level, or Debug breakpoints cannot be used in conjunction with AS COMMANDS.
- AS COMMANDS cannot be used with DETAIL.
    - When debugging accelerated programs, Debug may be used to set breakpoints at any TNS/R code address. As example 4 illustrates, such breakpoints are listed by printing TNS/R followed by the address in hex.
    - When listing all breakpoints with the AS COMMANDS option, breakpoints that were not set in the current Inspect session are not listed. When using this option to list a specific breakpoint, the following error is reported if the breakpoint was not set in the current Inspect session:

```
** Inspect error 350 ** Breakpoint was not set in this INSPECT session
```

Related Commands

- BREAK on page 6-19
- CLEAR on page 6-27
- SELECT DEBUGGER DEBUG on page 6-165

Output

The DETAIL clause results in the following output format:

```
Breakpoint Number: number

         Debugger:  Debug | Inspect
[            EVERY:  Every-limit [ (every-count) ]] (1)
[               IF:  condition                    ] (2)
[       Input text:  command line following BREAK ] (3)
         Location:  location
[          Option:  options | None                ] (4)
[            TEMP:  temp-remaining                ] (5)
[            THEN:  THEN action                   ] (6)
             Type:  Code | Data | ABEND | STOP
[          Subtype:  R/W | Write | Change            ] (7)
```

The notes indicate:

(1) Listed only for Inspect breakpoints with EVERY clause; every-count only shown if less than every-limit.

(2) Listed only for Inspect conditional breakpoints.

(3) Listed only for Inspect breakpoints.

(4) Listed only for Debug breakpoints; options is a comma-separated list of PRIV, CONDITIONAL, and ALL.

(5) Listed only for temporary Inspect breakpoints.

(6) Listed only for Inspect breakpoints with a THEN action.

(7) Listed only for Inspect breakpoints of type data.

Output from IF, Input Text, Location, and THEN can exceed one line. Data output is limited to a maximum of 512 characters. If there are more than 512 characters, the output will be truncated. For all the fields that can exceed one line, excess output will wrap as a hanging right paragraph, with line breaks indiscriminate with regard to data.

Examples

1. This is an example of possible LIST BREAKPOINT output:

```
Num Type Subtype Location
  1 Data Access  Byte Address %12 "DATA1"
  2 Data Access  Byte Address %14 "DATA2"
  3 Data Access  Byte Address %16 "DATA3"
```

2. This illustrates the default output of the LIST BREAKPOINT command. Note that data breakpoints have a subtype, which is one of: Change, Write, or R/W (Read/Write):

```
-PROGSC-LIST BREAKPOINT
Num Type Subtype Location
  1 Code    #ER^FATAL.#590
  2 Code    #ST^UNIT^HEADER.#4980 IF UNIT^OFFSET = 523222 THEN " t 1 arg"
  3 Data Change  Byte Address %1562 "CURRENT^PROGRAM^N"
  4 Code     #VA^BUILD^ADDRESS.#2012 EVERY 5
```

3. This illustrates detailed output for LIST BREAKPOINT DETAIL:

```
-PROGSC-LIST BREAKPOINT 3 DETAIL
Breakpoint Number: 3

      Debugger: INSPECT
    Input text: CURRENT^PROGRAM^N
      Location: Byte Address %1562 "CURRENT^PROGRAM^N"
          Type: Data
        Subtype: Change
```

4. This illustrates a TNS/R breakpoint set by Debug:

```
-PTALIN-LIST BREAKPOINT
Num Type Subtype Location
  1 Code DEBUG   TNS/R %h704201D0
```

5. This illustrates LIST BREAKPOINT DETAIL output for all breakpoints:

```
-PROGSC-LIST BREAKPOINT * DETAIL
Breakpoint Number: 1

      Debugger: INSPECT
    Input text: #ER^FATAL
      Location: #ER^FATAL.#590
          Type: Code

Breakpoint Number: 2

      Debugger: INSPECT
            IF: UNIT^OFFSET = 523222
    Input text: #ST^UNIT^HEADER IF UNIT^OFFSET = 523222 THEN " t 1 arg"
      Location: #ST^UNIT^HEADER.#4980
          THEN:  t 1 arg
          Type: Code

Breakpoint Number: 3

      Debugger: INSPECT
    Input text: CURRENT^PROGRAM^N
      Location: Byte Address %1562 "CURRENT^PROGRAM^N"
          Type: Data
       Subtype: Change
```

```
Breakpoint Number: 4

      Debugger: INSPECT
         EVERY: 5
    Input text: #VA^BUILD^ADDRESS EVERY 5
      Location: #VA^BUILD^ADDRESS.#2012
          Type: Code
```

6. This illustrates that Inspect always shows the breakpoint conditions first, followed by the breakpoint actions. For the two breakpoint conditions, EVERY and IF, EVERY is always shown first. For the breakpoint actions, THEN and TEMP, THEN is always shown first.

```
BREAK #PROC1 IF j = 17 EVERY 2                  -- conditions
BREAK #PROC2 TEMP 2 THEN "DISPLAY 'Hit proc2'"  -- actions
BREAK #PROC3 TEMP 2 THEN "DISPLAY 'Hit proc3'" IF j = 17 EVERY 2
```

```
Num Type Subtype Location
  1 Code         #PROC1.#10(FILE) EVERY 2 IF j = 17
  2 Code         #PROC2.#20(FILE) THEN "DISPLAY 'Hit proc2'" TEMP 2
  3 Code         #PROC3.#30(FILE) EVERY 2 IF j = 17 THEN "DISPLAY
                  'Hit proc3'" TEMP 2
```

LIST HISTORY

The LIST HISTORY command displays either a portion of or the entire history buffer.

```
LIST HISTORY [ command-line-range ] [ AS COMMAND[S] ]


command-line-range:   one of

  number [ / number ]
  search-text

search-text:   one of
    " [ character ]..."
    ' [ character ]...'
```

command-line-range

specifies a range of command lines to list.

number [/ *number*]

specifies the range of command-line numbers. If the number is positive, the specified number refers to an absolute command number. If the number is negative, the specified number is relative to the current command number.

`search-text`

is the most recent command line in the history buffer that begins with the text you specify. You need to specify only as many characters as necessary to identify the command line uniquely.

`AS COMMAND[S]`

directs LIST HISTORY to display the command lines as executable Inspect commands.

Default Value

If you do not specify a command range, LIST HISTORY displays the entire history buffer.

Related Commands

- FC on page 6-84
- HISTORY on page 6-93
- XC on page 6-224

LIST KEY

The LIST KEY command displays either one or all function-key definitions from the function-key list for the current Inspect session.

```
LIST KEY[S] [ key-name ] [ AS COMMAND[S] ]


key-name:   one of

   F1    F2    F3    F4    F5    F6    F7    F8
   F9    F10   F11   F12   F13   F14   F15   F16
   SF1   SF2   SF3   SF4   SF5   SF6   SF7   SF8
   SF9   SF10  SF11  SF12  SF13  SF14  SF15  SF16
```

`key-name`

specifies the function key whose definition you want to list. Valid function keys include F1 through F16 and shifted F1 (SF1) through shifted F16þ(SF16).

`AS COMMAND[S]`

directs LIST KEY to display the function-key definition as an ADD KEY command.

Default Value

If you do not specify a function key, LIST KEY displays all function-key definitions.

Related Commands

- ADD KEY on page 6-9
- DELETE KEY on page 6-30
- FK on page 6-90
- KEY on page 6-128

LIST PROGRAM

The LIST PROGRAM command displays the list of programs being debugged.

```
LIST PROGRAM[S] [ program | * [ , ] [ DETAIL ] ]


program: one of:

   program-number
   program-name
```

Usage Considerations

- An asterisk is displayed in front of the current program. If you are debugging a single program, it is always the current program. If you are debugging multiple programs, only one is the current program at any time.
- The system number is available in the DETAIL listing.
- The contents of the Type field are TNS, TNS SAVE, TNS/R, TNS/R SAVE, or PATHWAY. For example:

```
-PROGRAM-LIST PROGRAM
                  Program
Num  Program ID  Name       Type         State Location
 *1       13406  a.out      TNS/R        HOLD  #main.#5
  2       5,213  DEMO       TNS/R        RUN   #PROC.#12
```

- Location strings that would extend beyond the last column are broken at the last column and listed on the next line beginning at the starting column of the Location field.
- For PATHWAY requester programs, the PID field lists the PATHMON process name.

- If no program is specified, or "*" is specified, information is listed about all programs on the program list.

Related Commands

- ADD PROGRAM on page 6-10
- HOLD on page 6-93
- INFO OBJECTFILE on page 6-111
- INFO SAVEFILE on page 6-120
- PROGRAM on page 6-156
- SELECT PROGRAM on page 6-167
- STOP on page 6-215

Output

Use of the DETAIL clause results in the following output format:

```
-PROGRAM-LIST PROGRAM 1, DETAIL
Name: name
Number: number
               General Information
                CPU,PIN: cpu,pin
[               OSS PID: num                        ]  *1
       Guardian Version: tos version
        Instruction Set: TNS | TNS/R
               Location: current loc
              Processor: family (processor)
           Program File: filename
[             Libraries: library filename           ]  *2
[                        library filename           ]  *3
                  State: HOLD | STOP | GONE | RUN
                 System: name (number)
                   Type: TNS [SAVE] | TNS/R [SAVE] | PATHWAY

                 INSPECT Information

       ABEND Breakpoint: YES | NO
       Code Breakpoints: integer
       Data Breakpoints: integer
          Source System: None | name (number)
        STOP Breakpoint: YES | NO
```

- Fields labelled with a "*1" are listed only for OSS programs. Fields marked with a "*2" are listed only for programs that use libraries. A program may use zero or more libraries.
- Fields marked with a "*3" are listed only for TNS/R native programs, which can have multiple libraries.
- The processor will be listed as "unknown" if the program is a save file that was created prior to C30 or on D00.
- The processor will be listed as TNS/E for TNS programs on a TNS/E system.

- The instruction set will be listed as TNS/E for an OCA-accelerated program on a TNS/E system.
- Location strings and program names that would extend beyond the last column are broken at the last column and listed in the next line beginning at the starting column of the Location field, or the Program Name field respectively.

Examples

1. This example shows the LIST PROGRAM command without the DETAIL clause.

```
-DEMO-LIST PROGRAM
                        Program
Num Program ID Name       Type         State Location
   *1   09,00480 DEMO       TNS/R        HOLD  #main.#5.301(SDEMO)
    2   09,00490 DEMO       TNS/R SAVE   HOLD  #main.#5.301(SDEMO)
```

2. This example shows output for an OSS program with the DETAIL clause.

```
-PROGRAM-LIST PROGRAM 1, DETAIL
Name: a.out
Number: 1
               General Information


                 CPU,PIN: 8,160
                 OSS PID: 13406
        GUARDIAN Version: D2
         Instruction Set: TNS/R
                Location: #main.#5
                 Program: /usr/people/paul/bin/a.out
               Libraries: /usr/lib/libc.a
                          /usr/lib/libil8n.c
                   State: HOLD
                  System: \CUBS (175)
                    Type: TNS/R

                INSPECT Information

        ABEND Breakpoint: NO
        Code Breakpoints: 1
        Data Breakpoints: 0
           Source System: None
         STOP Breakpoint: NO
```

3. This example shows for a savefile of a TNS/R native program.

```
-DEMO-LIST PROGRAM 1, DETAIL
Name: DEMO
Number: 1
                  General Information

                Accelerated: NO
                    CPU,PIN: 2,33
           GUARDIAN Version: D40
            Instruction Set: TNS/R
                   Location: #main.#5.301(SDEMO)
                  Processor: TNS/R (NSR-L)
                    Program: \FOLK.$LORE.EXAMPLE.DEMO
                    Library: \FOLK.$SYSTEM.SYS01.ZCRESRL
                    Library: \FOLK.$SYSTEM.SYS01.ZCRTLSRL
                    Library: \FOLK.$SYSTEM.SYS01.ZOSSKSRL
                    Library: \FOLK.$SYSTEM.SYS01.ZI18NSRL
                    Library: \FOLK.$SYSTEM.SYS01.ZICNVSRL
                      State: HOLD
                     System: \FOLK (241)
                       Type: TNS/R SAVE

                 INSPECT Information

                  Save File: \FOLK.$LORE.HELP.DEMOS
                Source System: None
```

4. This example shows the DETAIL program listing for an accelerated TNS object file on a TNS/R system.

```
-CCLR-LIST PROGRAM CCLR DETAIL
Name: CCLR
Number: 1
              General Information

            Accelerated: YES
      Accelerated State: Register-exact
                CPU,PIN: 5,10
       GUARDIAN Version: G06
        Instruction Set: TNS/R
               Location: #CCL.#700(UMCCL)
              Processor: TNS/R (NSR-Y)
                Program: \SPEEDY.$SPIFF.TESTS.CCLR
                  State: HOLD
                 System: \SPEEDY (72)
                   Type: TNS

             INSPECT Information

       ABEND Breakpoint: NO
       Code Breakpoints: 0
       Data Breakpoints: 0
          Source System: None
        STOP Breakpoint: NO
-CCLR-
```

5. This example shows the DETAIL listing for an accelerated TNS object file on a TNS/E system. In this example, the program shown in the previous example has been processed by the TNS Object Code Accelerator (OCA).

```
-CCLE-LIST PROGRAM CCLE DETAIL
Name: CCLE
Number: 1
              General Information

            Accelerated: YES
      Accelerated State: Register-exact
                CPU,PIN: 5,7
       GUARDIAN Version: H06
        Instruction Set: TNS/E
               Location: #CCL.#700(UMCCL)
              Processor: TNS/E (NSE-P)
                Program: \PIPPIN.$D0117.KRIS.CCLE
                  State: HOLD
                 System: \PIPPIN (22)
                   Type: TNS

             INSPECT Information

       ABEND Breakpoint: NO
       Code Breakpoints: 0
       Data Breakpoints: 0
          Source System: None
        STOP Breakpoint: NO
-CCLE-
```

LIST SOURCE ASSIGN

The LIST SOURCE ASSIGN command displays the source assignments from the source-assignment list for the current Inspect session.

```
LIST SOURCE ASSIGN[S] [ AS COMMAND[S] ]
```

`AS COMMAND[S]`

directs LIST SOURCE ASSIGN to display the source assignments as ADD SOURCE ASSIGN commands.

Related Commands

- ADD SOURCE ASSIGN on page 6-14
- DELETE SOURCE ASSIGN on page 6-31
- SOURCE ASSIGN on page 6-202

LIST SOURCE OPEN

The LIST SOURCE OPEN command displays the names of the files that are currently open as a result of previous SOURCE commands.

```
LIST SOURCE OPEN[S]
```

Related Commands

- DELETE SOURCE OPEN on page 6-32
- SOURCE on page 6-196
- SOURCE OPEN on page 6-208

LOG

The LOG command records the session input, output, or both input and output in a permanent file.

```
LOG { [ BOTH | INPUT | OUTPUT  ] TO file-name }
    { STOP                                      }
```

`BOTH`

records both input commands and output response into the log file.

`INPUT`

records input commands to a log file.

`OUTPUT`

records output results from commands.

`file-name`

identifies a file to receive the copy of commands and output. If the file does not exist, a disk file is created with the name file-name.

Usage Considerations

- Logging is initiated when the command specifies a file name. If logging is already in progress, Inspect closes the previous LOG file and begins logging to the new file, unless the new file is the same as the previous LOG file. Inspect then ignores the LOG command.
- The file you specify can be a printer or a spooler collector.

- The current log file is closed and all logging is stopped when the LOG STOP command is entered or the current Inspect session is terminated.
- Inspect qualifies the log file using the current volume and subvolume if Inspect's systype is Guardian, otherwise, Inspect will qualify the file name using the current OSS directory.
- If a disk file is specified, and the file does not exist, Inspect will create a file (EDIT file if it is a Guardian systype, or a newline-terminated text file if it is an OSS systype). If the file already exists, Inspect appends the output to the file.

Related Commands

- ENV on page 6-81
- SYSTEM on page 6-217
- VOLUME on page 6-223

LOW

The LOW command switches Inspect from high-level command mode to low-level command mode. Section 7, Low-Level Inspect describes the low-level Inspect commands.

Note. The LOW command is invalid for PATHWAY requester programs.

```
LOW
```

Usage Considerations

- Automatic Command Mode Selection

  When Inspect receives a debug event, it sets the command mode to high-level or low-level, depending on the existence of symbol information for the scope unit in which the event occurred. When a symbol table is present, Inspect selects high-level mode; otherwise, it selects low-level mode.

- High-level Command Availability

  While in low-level, most high-level commands are available; however, abbreviations for high-level commands cannot be used while in low-level.

Usage Consideration for TNS/R Programs

- The usefulness of debugging accelerated programs at the TNS machine level is extremely limited. For more information, see Section 16, Using Inspect With Accelerated Programs on TNS/R Systems.

- Low-level Inspect is not supported for native programs. Use the SELECT DEBUGGER DEBUG command for TNS/R machine-level debugging.

Related Commands

- HIGH (low-level command)
- SELECT DEBUGGER DEBUG on page 6-165

MATCH

The MATCH command searches for scope-unit names or other identifiers in the current program.

```
MATCH { SCOPE pattern                                              }
      { IDENTIFIER pattern [[,] SCOPE scope-spec | [,VERBOSE]]}


scope-spec:   one of

   scope-ordinal
   scope-path
```

`SCOPE` *`pattern`*

directs Inspect to search the current program for scope-unit names that match the pattern you provide. The pattern must be at least one character long, and it can contain the wild-card characters ? and *. A question mark matches any single character; an asterisk matches any number of characters, including zero.

`IDENTIFIER` *`pattern`* `[ SCOPE` *`scope-spec`* `]`

directs Inspect to search for identifiers that match the pattern you provide. The pattern must be at least one character long, and it can contain the wild-card characters ? and *. A question mark matches any single character; an asterisk matches any number of characters, including zero.

`SCOPE scope-spec`

limits the identifier search to a single scope unit. You can specify the scope unit by its scope number (as shown by the TRACE command) or by its scope path.

If you omit this clause, Inspect first searches the current scope unit, then all containing scope units, and finally all global scope units.

`VERBOSE`

displays each scope searched, and any matching symbol.

Default Values

- If the MATCH IDENTIFIER command is entered with only one pattern, Inspect will display only the matching symbols, and the scopes they were found in.
- If the VERBOSE clause is omitted, scope names are displayed only if they have a matching pattern.

Usage Considerations

- Matching Uppercase and Lowercase Letters

  The MATCH command does not distinguish between uppercase and lowercase letters in the pattern unless the current language is C.
- Alias Restrictions with the MATCH command

  Aliases are not be expanded in the pattern with the MATCH IDENTIFIER or the MATCH SCOPE commands.
- Matching Scopes With a Leading "#"

  You must include a "#" to match scopes which have a leading "#". For example, to match the scope, "#global", enter:

```
MATCH SCOPE ##global
```

Related Commands

- INFO IDENTIFIER on page 105
- INFO SCOPE on page 6-122

Examples

1. This example searches for scope-unit names that begin with the characters IO^:

```
-TALOGJ-MATCH SCOPE io^*
Program Code:
IO^CLOSE
IO^OPEN
IO^READ
IO^WRITE
Program Data:
IO^DATA
```

2. This example searches for identifiers that contain the characters ERR:

```
-TALOBJ-MATCH IDENTIFIER *err*, VERBOSE
Searching ER^REPORT^ERROR
ERROR^NUMBER

Searching FLAGS^DATA^BLOCK
OVERRIDE^FLAG

Searching #GLOBAL
FILE^ERR
NEWPROCESS^ERROR
SPOOLER^ERROR
```

3. This example illustrates the VERBOSE clause of the MATCH IDENTIFIER command.

```
-PROGRAM-MATCH IDENTIFIER Global_Var, VERBOSE
Searching main
Searching Global^Block
  Global_var
Searching Proc^A
Searching Proc^B
Searching Proc^C...
```

4. This example illustrates the MATCH IDENTIFIER command without the VERBOSE clause.

```
-PROGRAM-MATCH IDENTIFIER Global_Var, VERBOSE
Found in scope Global^Block
   Global_Var
```

MODIFY

The MODIFY command changes the value of a data item or register in the current program. The current program must be in the hold state before you can use MODIFY. You can specify the new values in the MODIFY command, or you can let Inspect prompt you for them.

```
MODIFY { data-location [ WHOLE ] [ { = | := } mod-list ]  }
       { REGISTER register-name [ { = | := } expression ] }
       { SIGNAL signal-id [ { = | := } signal-list ] }


mod-list:

   mod-item [ , mod-item ]...


mod-item:

   [ integer COPIES ] expression
```

register-name: one of
tns-register-name
tns/r-register-name

tns-register-name: one of

```
   P    E    L    S
   R0   R1   R2   R3   R4   R5   R6   R7
   RA   RB   RC   RD   RE   RF   RG   RH
```

tns/r-register-name: one of

```
   $PC    $H1    $LO    $0    $1...$31
```
   tns/r-register-alias

tns/r-register-alias: one of

```
   $AT   $V0   $V1   $A0   $A1   $A2   $A3
   $S0   $S1   $S2   $S3   $S4   $S5   $S6   $S7
   $T0   $T1   $T2   $T3   $T4   $T5   $T6   $T7   $T8   $T9
   $K0   $K1   $GP   $SP   $FP   $RA
```

signal-id: one of:

```
   SIGABRT    SIGALRM   SIGFPE    SIGHUP   SIGILL     SIGINT
   SIGKILL    SIGPIPE   SIGQUIT   SIGSEGV  SIGTERM    SIGUSR1
   SIGUSR2    SIGCHLD   SIGCONT   SIGSTOP  SIGTSTP    SIGTTIN
   SIGTTOU    SIGABEND  SIGLIMIT  SIGSTK   SIGMEMMGR  SIGNOMEM
   SIGMEMERR  SIGTIMEOUT
```

signal-list:

signal-handler, mask, flags

signal-handler:

SIG_DEL | SIG_IGN | #SIG_DEBUG | #*function-name*

mask:

double [*double* [*double* [*double*]]]

flags:

double

`data-location [ WHOLE ] [ { = | := } mod-list ]`

modifies the data item specified by *`data-location`*. This data item must describe an area large enough to hold all values specified by *`mod-list`*. In addition, the data item cannot be a read-only array.

`WHOLE`

causes Inspect to treat the data item as a string of contiguous characters if it is a group item (a record or structure, for example).

The WHOLE clause is invalid for PATHWAY requester programs.

`mod-list`

specifies the new value or list of new values to be assigned to the data item. If *`mod-list`* specifies more than one value, *`data-location`* must specify a portion of an array (that is, an array name and subscript range).

`mod-item`

specifies a single new value or repeated instances of a single value. The syntax of *`mod-item`* is:

`[ integer COPIES ] expression`

`expression` specifies the new value, and the clause *integer* COPIES specifies the number of times to repeat the value.

`REGISTER register-name [ { = | := } expression ]`

modifies the register specified by register-name. Inspect assigns the value of *expression* to the given register.

TNS/R registers can only be modified when debugging a TNS/R program.

`register-name`

is the name of a TNS or TNS/R register.

`expression`

is an expression that yields the value that is to be assigned to the register.

`SIGNAL signal-id [ { = | := } signal-list ]`

modifies the signal specified by *`signal-id`*.

`signal-id`

identifies the signal. *`signal-id`* can be identified by letters or a number.

`signal-list`

is the value of the signal.

Default Value

If you do not supply a new value for the data item, register or signal, Inspect prompts you for it.

Usage Considerations

- Using the := and = Assignment Operators

  There is no difference between the two forms of the assignment operator. Either := or = can be used.

- Prompting for New Values

  When you modify a data item, Inspect normally prompts you for new values if the modifier list is not specified or does not contain enough values to fill the given data item. Inspect will not prompt if you use the WHOLE clause or if the MODIFY command is in an OBEY command file. In the latter case, if the new values do not completely fill the data item, the remainder is left unchanged.

  Inspect prompts you by displaying each element of the data item with its name and current value. Entering a value modifies the element, and entering a comma retains the current value. Inspect continues prompting until the last element of the data item is displayed, or until you enter only a RETURN (indicating that there are no further modifications).

- Modifying Registers

  If no value is specified, the current value of the register is displayed in the current output radix and a prompt is issued for a new value.

- Modifying Strings

  When you modify a string variable, the new string cannot exceed 250 bytes. If it does, Inspect displays this warning message:

```
** Inspect warning 95 ** Maximum string length (250 bytes) is exceeded.
                         String is truncated.
```

  Inspect changes only the first 250 characters of the string. To avoid this restriction, modify the string in parts:

```
-PRG-MODIFY string(1:100) = "100 characters..."
-PRG-MODIFY string(101:200) = "100 more characters..."
-PRG-MODIFY string(201:300) = "another 100 characters..."
```

- Modifying Signals

  There are three pieces of information associated with a signal: the signal handler, the mask, and a flags word. Inspect prompts you for the values of the mask and flags word if they have been omitted. The mask is composed of 128-bits split into 4-32 bit double words.

- Modifying FILLER Elements of a Record

  Using the WHOLE clause enables you to assign values to all elements of a record, including those designated FILLER.

- Interactive MODIFY in a Command List

  If you include an interactive MODIFY command (one where Inspect needs to prompt you for values) in a command list, it must be the last command in the command list.

Usage Considerations for Accelerated Programs

- When debugging an accelerated program on a TNS/R system, values may be cached in machine registers when the current location is not a register-exact point. In such an event, the modify operation may have no effect.
- TNS/R registers can only be modified when debugging an accelerated or native program on a TNS/R system. An error is reported if you attempt to modify registers on a non-accelerated program.
- When debugging accelerated programs, TNS registers can only be modified at register-exact points. For more information, see Section 15, Using Inspect on a TNS/R System.
- When debugging accelerated programs, the P register can only be set to a value that is the address of a register-exact point. For more information, see RESUME on page 6-158.

Related Commands

- DISPLAY on page 6-33
- INFO IDENTIFIER on page 6-105

Examples

1. This example demonstrates how various MODIFY commands affect this COBOL data structure:

```
01 B.
  05  C PICTURE X(10) VALUE IS "Jelvo, B. ".
  05  D PICTURE S99 USAGE IS COMP VALUE IS 45.
  05  E PICTURE 9(10) VALUE IS 4085551212.
  05  F PICTURE X(24) VALUE IS "Alvin's Place".
```

Before making modifications, display B field-by-field and then as a single string:

```
-COBOLOBJ-DISPLAY B
B =
  C = "Jelvo, B. "
  D = 45
  E =  4085551212.
  F = "Alvin's Place
                            "
-COBOLOBJ-DISPLAY B WHOLE
B = "Jelvo, B. " ?0 "-4085551212Alvin's Place           "
```

Change the C field and then DISPLAY to verify the change:

```
-COBOLOBJ-MODIFY C OF B = "Manley, G."
-COBOLOBJ-DISPLAY B
B =
  C = "Manley, G."
  D = 45
  E =  4085551212.
  F = "Alvin's Place           "
```

Try a MODIFY WHOLE, and then DISPLAY to verify the changes:

```
-COBOLOBJ-MODIFY B WHOLE = "Wirble, U.--6662325468Willy"
-COBOLOBJ-DISPLAY B WHOLE
B = "Wirble, U.--6662325468Willy's Place           "
-COBOLOBJ-DISPLAY B
B =
  C = "Wirble, U."
  D = 11565
  E =  6662325468.
  F = "Willy's Place           "
```

Use the & operator to concatenate ASCII and Non-ASCII values in the alphanumeric data item C and then DISPLAY to verify the changes:

```
-COBOLOBJ-DISPLAY C of B
B.C = "Jelvo, B. "
-COBOLOBJ-MODIFY C of B
B.C = "Jelvo, B. " := "Paul,"&%hff&%hff&"D. "
-COBOLOBJ-DISPLAY C of B
B.C = "Paul," ?255 ?255 "D. "
```

DISPLAY D in several bases:

```
-COBOLOBJ-DISPLAY D IN OCTAL DECIMAL HEX ASCII
B.D = %26455 11565 %H2D2D "--"
```

2. This example demonstrates various MODIFY commands applied to these FORTRAN data structures:

```
REAL A
CHARACTER*6 S
```

```
INTEGER K(10,10)
INTEGER ARR (12)
```

```
-FORTOBJ-MODIFY A=5
-FORTOBJ-MODIFY S="Falcon"
-FORTOBJ-MODIFY K(1:5,3) := 4,7,9,15,22
-FORTOBJ-MODIFY K(1:5,4) := 5 COPIES 0
-FORTOBJ-DISPLAY A, S, K(1:5, 3:4)
A = 5, S = "Falcon"
K[1,3] = 4 7 9 15 22
K[1,4] = 0 0 0 0 0
```

Use Inspect to prompt for new values:

```
-FORTOBJ-MODIFY ARR(6:11)
  ARR(6)=49 := ,
  ARR(7)=50 := 37
  ARR(8)=51 :=
```

The comma above leaves ARR(6) unchanged, and pressing carriage return aborts the Inspect prompting, leaving ARR(8) through ARR(11) unchanged.

OBEY

The OBEY command causes Inspect to read commands from a specified file. A file named in an OBEY command is called an OBEY file.

```
OBEY filename
```

filename

is the file name to obey.

Usage Considerations

- Inspect reads and processes commands from the named file until it encounters the end-of-file. At this point, Inspect closes the OBEY file and reverts to its previous input file, normally the Inspect command terminal.
- Additional OBEY commands can appear within an OBEY file; OBEY files can be nested to a depth of four.
- Inspect qualifies the log file using the current volume and subvolume if Inspect's systype is Guardian, otherwise, Inspect will qualify the file name using the current OSS directory.
- Inspect generates an error if any part of the specification is invalid, if the file does not exist, or if the file cannot be opened. Inspect displays an error message and prompts for input if the input file is a terminal. If the input file was not a terminal, Inspect terminates.
- If you use an OBEY command in the THEN clause of a BREAK command, an error can occur if the OBEY file contains a RESUME command. The first time the

breakpoint is activated, Inspect opens and reads commands from the OBEY file. When Inspect encounters the RESUME command in the OBEY file, it lets the program resume execution but does not close the OBEY file. The next time the breakpoint is activated, Inspect will again try to open the OBEY file. However, the OBEY file is still open as a result of the first breakpoint activation. Consequently, Inspect reports the error:

```
** Inspect error 14 ** Illegal OBEY file - ignored
```

- You should not use abbreviations of Inspect keywords in an OBEY file because the abbreviations might change in future releases of Inspect.

Related Commands

- OUT on page 6-155
- SYSTEM on page 6-217
- TERM on page 6-218
- VOLUME on page 6-223

OBJECT

The OBJECT command displays information about the current program's object file. The OBJECT command is a synonym for the INFO OBJECTFILE command.

Note. The OBJECT command is invalid for PATHWAY requester programs.

Related Commands

- INFO OBJECTFILE on page 6-111
- INFO SAVEFILE on page 6-120

OPENS

The OPENS command shows the status of files that have been opened by the current program. The OPENS command is a synonym for the INFO OPENS command.

Note. The OPENS command is invalid for PATHWAY requester programs.

```
OPENS  [ { * | file-list  } [ DETAIL ] [ file-type ] ]

file-list:

   file-number  [ , file-number ]...

file-type:   one of

   FORTRAN   FD   GUARDIAN
```

`*`

requests the status of all files opened by the current program.

`file-list`

requests the status of specific files. *`file-list`* is a list of file numbers identifying the desired files.

`file-number`

specifies a single file. *`file-number`* can be:

A COBOL FD name
An expression that evaluates to an integer value
A data location identifying an integer value

Inspect interprets the integer values as file-system file numbers, unless you specify the F clause; Inspect would then interpret the values as FORTRAN logical-unit numbers.   If a FD clause is used, Inspect interprets the values as OSS file descriptors.

`DETAIL`

directs Inspect to display the maximum available information for the specified files.

`file-type`

indicates to Inspect what type of file to display. *`file-type`* can be one of the following:

`FORTRAN`

indicates that *`file-number`* values specify FORTRAN logical-unit numbers, not file system file numbers. Files without FORTRAN logical units will not be displayed.

`FD`

indicates that *`file-number`* values specify OSS file descriptors. Files without OSS file descriptors will not be displayed.

`GUARDIAN`

indicates that *`file-number`* values specify Guardian system file numbers. Files without Guardian file system numbers will not be displayed.

Default Values

- The default file type is Guardian file system numbers.
- If you enter the OPENS command without parameters, Inspect displays the status of all files opened by the current program.

OUT

The OUT command directs the output listing to a specified file.

```
{ OUT filename   }
{ /OUT filename/ }
```

`filename`

is a file name.

Usage Considerations

- The first form of the OUT command causes permanent redirection of the output.
- The second form of the OUT command causes temporary redirection of the output. This form is specified as part of another command. You must enter it immediately after the other command name and before any other part of that command; for example:

  ```
  HELP /OUT newuser/ NEW-USER
  ```

- If *filename* specifies a disk file and the file does not exist, an EDIT file is created. If the named file is an existing disk file, the output is appended to the file.
- If the file name is invalid or if the file cannot be opened, an error occurs. An error message is displayed and the command is not executed.
- Inspect qualifies *filename* using the current volume and subvolume of the Inspect session. If you have not set these explicitly, they are the default volume and subvolume of the Inspect session. You can use the ENV command to determine the current defaults.

Related Commands

- OBEY on page 6-152
- SYSTEM on page 6-217

- TERM on page 6-218
- VOLUME on page 6-223

PAUSE

The PAUSE command suppresses Inspect prompts until a debug event occurs in any of the programs on the program list.

```
PAUSE
```

Usage Considerations

- Inspect begins prompting after a PAUSE when:
  - You press the BREAK key, assuming that the program has not taken control of the BREAK key.
  - Any running program under the control of Inspect generates a debug event. (see the information on debug events and execution states in Section 2, Inspect Concepts).
  - Inspect receives a wake-up message from any process. For more information, see the *TACL Programmer's Guide.*
- Using PAUSE to Share the Inspect Command Terminal

  If Inspect and the program you are debugging share the same terminal, you can use the PAUSE command to return control of the terminal to your program.

PROGRAM

The PROGRAM command displays the programs in the program list or selects a program as the current program.

```
PROGRAM[S] [ program [ CODE code-file ]
                     [ LIB lib-file ]
                     [ SRL {(srl-file [ , srl-file,...])}]


program:   one of

   program-number
   program-name
   cpu,pin
```

program

specifies a program using one of several formats. *Program-number* identifies a program by its program number (as shown by the LIST PROGRAM command).

`program-name` identifies a program by its program name (as shown by the LIST PROGRAM command). `cpu,pin` identifies a process by its process ID (CPU number and process number).

`CODE` *`code-file`*

directs Inspect to retrieve symbol information from an object code file different from the one used to create the process (or the process that was saved).

`LIB` *`lib-file`*

directs Inspect to retrieve symbol information from a library file different from the one associated with the process (or the process that was saved).

`SRL` *`srl-file`*

directs Inspect to retrieve symbol information from a SRL file different from the one associated with the process (or the process that was saved). The SRL clause is a single filename, or comma separated list of filenames.

Default Value

If you do not specify *program*, PROGRAM displays all programs in the current sessions.

Usage Considerations

- If you specify *`program`*with optional CODE, LIB, or SRL clauses, PROGRAM selects the named program from the program list as the current program.
- The PROGRAM command will not add the program if *`program`* is not currently on the program list.
- You cannot select programs using an OSS PID. Use the *`cpu,pin`* of the process or the program ordinal.
- The file name type of the CODE, LIB, or SRL clauses is determined by the current systype of Inspect.

Related Commands

- ADD PROGRAM on page 6-10
- LIST PROGRAM on page 6-137
- SELECT PROGRAM on page 6-167
- STOP on page 6-215

RESUME

The RESUME command reactivates a suspended program, changing the program's state from hold to run. The RESUME command is not valid for save files.

```
RESUME [ * [ EXIT ]                                        ]
       [ program ] [ AT code-location [ , RP integer ] ]


program:   one of

   program-number
   program-name
   cpu,pin
   PATHWAY-terminal-name
```

`* [ EXIT ]`

resumes all executable programs currently in the hold state.

The optional EXIT clause directs Inspect to clear all breakpoints in all programs on the program list and then terminate the Inspect session after it resumes the programs.

`[ program ] [ AT code-location [ , RP integer ] ]`

resumes a single executable program. You can specify the program using one of several formats. *program-number* identifies the program by its program number (as shown by the LIST PROGRAM command). *program-name* identifies the program by its program name (as shown by the LIST PROGRAM command). *cpu,pin* identifies a process by its process ID. PATHWAY-*term-name* identifies a PATHWAY requester program by the name of its logical PATHWAY terminal.

The AT clause specifies that execution is to resume at a location other than the next sequential one. The code location you provide in the AT clause must represent the location of an executable instruction within the current scope unit. Using the AT clause might require that you adjust the register pointer (RP) field of the E-register before execution resumes.

The RP clause specifies the value for the RP field of the E-register. The integer you provide in the RP clause must be in the range 0 to 7.

You should not use the RP clause unless you understand the underlying machine structure, particularly the relationship of the RP field to instruction execution. For more information, see the *System Description Manual*.

The AT and RP clauses are invalid for PATHWAY requester programs, and the RP clause is invalid for accelerated programs.

Default Value

Entering RESUME without parameters resumes the execution of the current program.

Usage Considerations

- RESUME in Command Lists

  In a command list, RESUME must be the last command.

- RESUME in OBEY Files

  If you use RESUME in an OBEY file and then refer to that OBEY file in the THEN clause of the BREAK command, Inspect reports an error. For more information, see the description of the OBEY on page 6-152.

- Resuming by Program Name

  If you are debugging more than one process with the same program name, Inspect resumes the oldest process when you use the *program-name* form of *program*. To override, use the *program-number* or *cpu,pin* form of *program*.

Usage Considerations for Accelerated Programs

- When debugging accelerated programs on a TNS/R system, it is not possible to transfer program execution to arbitrary program locations as it is on the TNS. The current location must be a register-exact point, the destination must also be a register-exact point, and the value of RP must be the same at both locations. If the target location is not a register-exact point, this error is reported:

```
** Inspect error 357 ** Target location must be a register-exact point
```

  If your current location is not a register-exact point, this error is reported:

```
** Inspect error 370 ** Current location must be a register-exact point
```

Note. The register-pointer (RP) clause of the RESUME command is not supported for accelerated programs.

- The effect of the RESUME AT command can also be achieved by using either the MODIFY REGISTER command or the low-level M command to modify the value of the P register. As described previously, register modification commands are restricted to operate only at register-exact points. When the operand is the P register, they are further restricted to accept only values that are the address of register-exact points. This error is reported if the value is not the address of a register-exact point:

```
** Inspect error 356 ** Value must be the address of a register-exact point
```

- It is not possible for Inspect to determine the value of RP at the source and destination locations. You must therefore ensure that the destination location has the same RP value when changing the value of the P register. In general, the RP value usually corresponds to the return value size at return points, and is 7 at other register-exact points.

Related Commands

- HOLD on page 6-93
- STEP on page 6-212
- STOP on page 6-215

SAVE

The SAVE command creates a save file of the current program. A save file is a “snapshot” or “image” of a process or PATHWAY server stored on disk. A save file contains information regarding the state of the process or PATHWAY server at the moment the image was created. This information includes:

- The user data space (including extended data segments).
- The values of TNS machine registers.
- The values of TNS/R machine registers for accelerated programs or native programs.
- The names and status of any files opened by the process or PATHWAY server.
- General information, including: the TOS version, the processor type, the last debugging event, the user ID and name of the creator, and the creation timestamp.

Note. The SAVE command is invalid for Pathway requester programs.

```
SAVE filename [ ! ]
```

filename

specifies the name of the save file.

!

specifies that if a file named file-name already exists, it must be replaced by the new save file. If you omit the exclamation point, an attempt to use an existing file produces a file system error 10 and leaves the old file intact.

Usage Considerations

- All save files created by Inspect are of type unstructured and have a file code ofþ130.
- Inspect qualifies filename using the current subvolume if systype is Guardian or the current working directory if systype is OSS. If you have not set the volume and subvolume explicitly, they are the default volume and subvolume of the Inspect session. Use the ENV command to determine these defaults.
- If the only program you are debugging is a save file, entering the STOP command closes the save file but does not end the Inspect session, as it does for processes. In this case, you must enter the EXIT command or press CTRL/Y.
- To begin analyzing a save file, enter an ADD PROGRAM command with the save file name. You can use any high-level or low-level command that does not change the execution state of the process; therefore, you can display code values, data values, attributes, and a stack trace. Many commands that are available when a process is in the hold state, however, are unavailable when you are working with a save file. These commands include:

  BREAK
  CLEAR
  FB
  HOLD
  LIST BREAKPOINT
  MODIFY
  RESUME
  SELECT DEBUGGER DEBUG
  STEP

Related Commands

- ADD PROGRAM on page 6-10
- INFO SAVEFILE on page 6-120
- STOP on page 6-215

SCOPE

The SCOPE command changes or displays the current scope path. For more information about scopes, see Scope Paths on page 2-12.

```
SCOPE [ scope-spec ]


scope-spec:   one of

    scope-number
    scope-path [ (instance) ]
    #data-block
    ##GLOBAL
```

scope-spec

specifies the scope path that Inspect is to use as the current scope path when qualifying unqualified identifiers.

scope-number

specifies the path to an active scope unit by the unit's scope number (as displayed by the TRACE command).

scope-path [(*instance*)]

specifies a named scope path as the current scope path. You can include an instance to differentiate activations of the same scope unit.

The (*instance*) option is invalid for COBOL programs.

#data-block

specifies a named data block as the current scope path.

##GLOBAL

specifies the implicitly named global data block in TAL as the current scope path.

Default Value

If you enter the SCOPE command without any parameters, Inspect displays the current scope path.

Usage Considerations

- To examine an element as it exists during a particular instance, include an instance number when you specify the scope path. You can count from either direction using these conventions:
  - Instance 1 is the least recent instance—that is, the oldest chronologically. Positive values count from the base of the stack toward the top.
  - Instance 0 is the most recent instance—that is, the current scope path.
  - Instance -1 is the next most recent—that is, the youngest chronologically. Negative values count from the top of the stack toward the base.
- There is a subtle difference between an unspecified instance and instanceþ0. Suppose you have a scope unit named BINK. You can set your scope path to #BINK whether BINK is active or not; you can set your scope path to #BINK(0) only if BINK is active. If you set your scope path to #BINK and the scope unit is active, then #BINK(0) and #BINK are identical.
- If the current program is not in the hold state when you enter a SCOPE command without parameters, Inspect displays the scope path that was active at the time the command was received.
- If you set your scope path to a data block, you cannot set unqualified code breakpoints.
- Whenever a debug event occurs, Inspect sets the current scope path to the path that specifies the scope unit containing the location at which the event occurred.

Related Commands

- ENV on page 6-81
- INFO SCOPE on page 6-122
- MATCH with the SCOPE option on page 6-144
- TRACE on page 6-219

SELECT

This diagram shows the complete syntax for the SELECT command and its clauses. Detailed descriptions of the clauses, including usage considerations and examples, are presented in the following subsections.

```
SELECT select-option


select-option:   one of

   DEBUGGER DEBUG
   LANGUAGE language
   PROGRAM program [ CODE file-name ]
                    [ LIB file-name ]
                   [ SRL {(srl-file [ , srl-file,...])}]
   SEGMENT [ segment id ]
   SOURCE SYSTEM [ \system ]
   SYSTYPE { GUARDIAN | OSS }


language:   one of
   C         C++      COBOL     COBOL85
      FORTRAN   Pascal   SCOBOL    TAL


program:   one of

   program-number
   program-name
   cpu,pin
```

Usage Consideration

Anything you select may be overridden by Inspect when a subsequent event is received. The SELECT command differs from the SET command, which maintains its setting until you change it.

Related Commands

- ADD on page 6-6
- ENV on page 6-81
- LIST on page 6-129

SELECT DEBUGGER DEBUG

The SELECT DEBUGGER DEBUG command allows you to invoke an alternate debugger:

- On a TNS/R system, Debug is the alternate debugger for TNS or TNS/R programs. You interact with Debug until issuing the Debug command “INSPECT”, which returns control of the program to Inspect.
- On a TNS/E system, Native Inspect is the alternate debugger for TNS/E native programs (Native Inspect replaces Debug as the system debugger on TNS/E systems). You interact with Native Inspect until you issue the Native Inspect switch command, which returns control of the program to Inspect.

  TNS programs running on a TNS/E system can only be debugged using Inspect.

```
SELECT DEBUGGER DEBUG
```

General Usage Considerations

- When you return to Inspect from Debug, the Inspect breakpoint list is updated to reflect breakpoints that have been set in Debug.

  If the addition of breakpoints to the Inspect breakpoint list would cause the number of breakpoints to exceed the Inspect limit on numbered breakpoints (99), breakpoints that are added are not assigned an ordinal number.
- Debug can only be invoked on a process or PATHWAY server that is in the HOLD state.
- Conditional breakpoints set by Inspect are reported unconditionally while in Debug.
- Conditional breakpoints set by Debug are always evaluated conditionally, regardless of whether Inspect or Debug is being used.
- When Debug is invoked from within Inspect and the process terminates, control returns to Inspect. Inspect terminates if there are no other processes being debugged; otherwise, Inspect resets the current program and issues a prompt.
- When Debug is invoked from an Inspect being used to debug multiple processes, Inspect waits either for a debugging event to be reported for another process or for the control to return from Debug. If an event is received, Inspect reports the event and issues a prompt. This may cause Inspect and Debug to compete for control of the terminal. The Inspect PAUSE and Debug P commands can be useful in this case.
- If Inspect is used to set breakpoints on STOP and/or ABEND, the breakpoints are reported by Inspect, even if the process calls STOP and/or ABEND while Debug is being used to debug it.

- In some debugging situations, the differences between Inspect and Debug may be important. Inspect runs as an application process while Debug actually runs as part of the process being debugged. As a result, Inspect may affect the state of the system more than Debug. Debug, on the other hand, makes use of the program's stack, which can affect program behavior.
- If the breakpoint has attributes which are unique to Debug, such as conditional, the breakpoint is listed as either type “Code DEBUG” or type “Data DEBUG”. The text field of such breakpoints lists the breakpoint's attributes. These tokens may be listed:

```
CONDITIONAL     - The breakpoint is Debug conditional
ALL PROCESSES   - The breakpoint applies to all processes
PRIV            - The breakpoint was set by PRIV Debug
```

Usage Considerations for Privileged Users

- Care must be exercised when returning control to Inspect when Debug has been used to set “all process” breakpoints in System Code and System Library spaces. A deadlock can occur if Inspect inadvertently calls the procedure in which the breakpoint is set.
- Privileged breakpoints are added to the Inspect breakpoint list only if PRIV MODE is set ON.
- Debug does not retain any state information between invocations from Inspect. As a result, privileged Debug mode is reset when control is returned to Inspect; the Debug PRV command must be reissued after re-entering Debug. Inspect on the other hand does retain state information when switching to and from Debug.

Usage Consideration for Accelerated Programs

- When debugging programs on a TNS/R system, Debug can be used to debug at the TNS/R machine level.

Related Commands

- INFO on page 6-104
- LIST BREAKPOINT on page 6-131

SELECT LANGUAGE

The SELECT LANGUAGE command changes the current source language. The current source language affects the acceptable syntax of language-dependent entities such as code locations, data locations, and expressions.

Inspect provides the SELECT LANGUAGE command to aid in debugging a program compiled from more than one source language. SELECT LANGUAGE enables you to switch between language-dependent syntaxes; therefore, you can avoid incompatibilities between languages. The difference between the use of the caret (^) in TAL and Pascal is an example of a language incompatibility.

```
SELECT LANGUAGE language


language:   one of

   C          C++       COBOL     COBOL85
   FORTRAN    Pascal    SCOBOL    TAL
```

language

specifies which source language Inspect should make current.

Usage Considerations

- Automatic Selection of the Current Language

  Whenever a debug event occurs, Inspect sets the current language to the language specified by the language attribute of the scope unit containing the location at which the event occurred.

- Mixing C with other languages

  If the current language is C, Inspect will not upshift identifiers, otherwise, Inspect upshifts all program identifiers and keywords.

Related Command

ENV

SELECT PROGRAM

The SELECT PROGRAM command selects a program from the program list as the current program.

```
SELECT PROGRAM program [ CODE code-file ]
                       [ LIB lib-file ]
                       [ SRL {(srl-file [ , srl-file,...])}]


program:   one of

   program-number
   program-name
   cpu,pin
```

`program`

specifies a program using one of several formats. *`program-number`* identifies a program by its program number (as shown by the LIST PROGRAM command). *`program-name`* identifies a program by its program name (as shown by the LIST PROGRAM command). *`cpu,pin`* identifies a process by its process ID (CPU number and process number).

`CODE` *`code-file`*

directs Inspect to retrieve symbol information from an object code file different from the one used to create the process (or the process that was saved). Cannot be used to specify a TNS/E native object file.

`LIB` *`lib-file`*

directs Inspect to retrieve symbol information from a library file different from the one associated with the process (or the process that was saved). Cannot be used to specify a TNS/E native object file.

`SRL` *`srl-file`*

directs Inspect to retrieve symbol information from a SRL file different from the one associated with the process (or the process that was saved). The SRL clause is a single filename, or comma separated list of filenames.

Usage Consideration

- Using the CODE and LIB Clauses

  Inspect provides the CODE and LIB clauses so that you can obtain symbol information even when the object or library file does not contain symbols. For example, most applications include symbols only during their development—the symbol information is stripped out before distribution. If a bug is then discovered, the customer can make a save file and return it to the developers. Using the CODE and LIB clauses, the developers can then associate their versions of the object and libraries (with symbols) to the save file, therefore enabling them to use high-level, symbolic Inspect to locate the problem more quickly.

  On a TNS/E system, the CODE and LIB clauses cannot be used to specify a TNS/E native object file. If you do so, Inspect displays an error message (“Inspect cannot read a TNS/E object file”).

Related Commands

- ADD PROGRAM on page 6-10
- LIST PROGRAM on page 6-137
- PROGRAM on page 6-156
- STOP on page 6-215

SELECT SEGMENT

The SELECT SEGMENT command selects extended data segments in which extended data addresses are to be resolved. In effect, this controls the extended data segment viewed by Inspect.

```
SELECT SEGMENT[S] [ segment-id ]
```

segment-id

is the ID of an extended segment. Any selectable segment ID listed by the INFO SEGMENTS command is valid. If *segment-id* is not specified, the extended data segment used by Inspect is that which is currently in use by the program.

Usage Considerations

- If the extended data segment selected by Inspect is different than the process's current extended segment, the segment is marked with a “>“ in the output of the INFO SEGMENTS command.
- Although any flat data segment can be displayed using the INFO SEGMENTS command, selecting the segment using the SELECT SEGMENT command will result in this error:

```
** Inspect error 70 **  Specified segment can not be selected
```

Related Command

INFO SEGMENTS

SELECT SOURCE SYSTEM

The SELECT SOURCE SYSTEM command directs Inspect to retrieve source files from a specific system when the object file has moved but the source has not.

```
SELECT SOURCE SYSTEM [ \system ]
```

system

specifies the system from which Inspect should retrieve source files.

Default Value

If you do not specify a system name in a SELECT SOURCE SYSTEM command, Inspect uses the name of your current system.

Usage Considerations

- Inspect provides the SELECT SOURCE SYSTEM command for one common context in particular: when a program's object code is copied to another system, but its source code is not.
- SELECT SOURCE SYSTEM sets the source system for the current program. Each program that you are debugging can have its own source system.
- Inspect determines whether it should add the system name specified by SELECT SOURCE SYSTEM after it formulates the current name (by applying any matching source assignment to the compilation name).
- The SELECT SOURCE SYSTEM command cannot be used if the current name of a source file in the current program explicitly includes a system name. Use the ADD SOURCE ASSIGN command instead.
- If you used a C-series compiler and your object file has been moved to another node on a network and the source files are at the same location as when the object file was compiled, use the SELECT SOURCE SYSTEM command to identify the node that the files reside on.
- To determine the source system for the current program, use the ENV command.

Related Commands

- ADD SOURCE ASSIGN on page 6-14
- ENV on page 6-81
- SOURCE on page 6-196
- SOURCE SYSTEM on page 6-211

SELECT SYSTYPE

The SELECT SYSTYPE command allows you to change the current systype of Inspect.

```
SELECT SYSTYPE { GUARDIAN | OSS }
```

Usage Consideration

Inspect may override any user selected systype when a subsequent event is received.

SET

The SET command controls Inspect parameters that enable you to customize your Inspect session.

This is the complete syntax for the SET command and its clauses. Detailed descriptions of the clauses, including usage considerations and examples, are presented in the following subsections.

```
SET set-assignment


set-assignment:   one of

   CHARACTER FORMAT [=] [ ASCII | XASCII | GRAPHIC[S] ]
   DEREFERENCE DEPTH [=] integer
   ECHO echo-item [=] { ON | OFF }
   HELP FILE [=] filename
   LOCATION FORMAT [ level ] [=] loc-fmt [, loc-fmt ]...
   PRIV MODE [=] ON | OFF
   PROMPT [=] [ prompt-item [ , prompt-item ]... ]
   RADIX [ INPUT | OUTPUT ] [ level ] [=] radix
   SOURCE BACK [=] count
   SOURCE FOR [=] count
   SOURCE RANGE [=] range / range
   SOURCE WRAP [=] { ON | OFF }
   STATUS ACTION [ level ] [=] [ cmd-string ]
   STATUS LINE25 [ level ] [=] [ status-item-list ]
   STATUS SCROLL [ level ] [=] [ status-item-list ]
   SUBPROC SCOPING [=] [ SUBLOCAL | LOCAL ]
   SYSTYPE { GUARDIAN | OSS }
   TRACE trace-level [=] { ON | OFF }


echo-item:  one o

   ALIAS[ES]   HISTORY   KEY[S]


level:   one of

   BOTH   HIGH   LOW


loc-fmt:   one of

   INSTRUCTION[S]
   LINE[S] [ FILE [ ALL ] ] [ OFFSET ]
   STATEMENT[S] [ OFFSET ]
```

```
prompt-item:   one of

   string          ACCELERATOR STATE     COMMAND
   DIRECTORY       ICODE                 INSTRUCTION SET
   LEVEL           FN                    PROCESSOR
   PROGRAM FILE    PROGRAM NAME          PROGRAM ORDINAL
   PROGRAM PID     RADIX                 SOURCE
   STEP            SUBVOL[UME]           SYSTEM
   SYSTYPE         VOLUME


radix:   one of

   DEC[IMAL]   HEX[ADECIMAL]   OCT[AL]


count:

   integer [ STATEMENT[S] | LINE[S] | INSTRUCTION[S] ]


range:   one of

   F   L   #line-number   statement-number


status-item-list:

   status-item [ , status-item ]...


status-item:   one of

   string            ACCELERATOR STATE  EVENT
   INSTRUCTION SET   LANGUAGE           LOCATION
   PROCESSOR         NEW LINE           PROGRAM FILE
   PROGRAM NAME      PROGRAM ORDINAL    PROGRAM PID
   SCOPE             STATE              SYSTYPE
   TYPE


trace-level:   one of

   ARGUMENT[S]   SCOPE[S]   STATEMENT[S]
```

SET CHARACTER FORMAT

The SET CHARACTER FORMAT command changes the default output format for character identifiers.

```
SET CHARACTER FORMAT [=] [ ASCII | XASCII | GRAPHIC[S] ]
```

`ASCII`

displays items as printable 7-bit ASCII characters. If a byte value is not 7-bit ASCII (32-127), it is shown as ?nnn, in the current output radix.

`XASCII`

displays items as printable 8-bit extended ASCII characters. If a byte value is not 8-bit ASCII (32-255), it is shown as ?nnn, in the current output radix.

`GRAPHICS`

displays items as characters, including control characters.

Default Value

The default INSPLOCL file sets the character format as follows:

```
SET CHARACTER FORMAT = ASCII
```

Related Commands

- DISPLAY on page 6-33
- SET RADIX on page 6-181
- SHOW command with CHARACTER FORMAT option on page 6-194

SET DEREFERENCE DEPTH

The SET DEREFERENCE DEPTH command allows you to control automatic dereferencing within TAL STRUCTs.

```
SET DEREFERENCE DEPTH [=] integer
```

`integer`

specifies how many times Inspect should recursively dereference a pointer within a TAL structure.

Default Value

The default INSPLOCL file sets the dereference depth as follows:

```
SET DEREFERENCE DEPTH = 0
```

Related Commands

- DISPLAY on page 6-33
- SHOW command with DEREFERENCE DEPTH option on page 6-194

SET ECHO

The SET ECHO command controls whether Inspect redisplays, or echoes, command lines when you use an alias, a function key, or the XC command.

```
SET ECHO echo-item [=] { ON | OFF }


echo-item:   one of

   ALIAS[ES]   HISTORY   KEY[S]
```

echo-item

specifies which echo control you want to set. ALIAS controls echoing of command lines containing aliases. HISTORY controls echoing of command lines reissued by the XC command. KEY controls echoing of command lines entered by pressing a function key.

Default Value

The default INSPLOCL file sets the echo controls as follows:

```
SET ECHO ALIASES = OFF
SET ECHO HISTORY = ON
SET ECHO KEYS = OFF
```

Related Command

SHOW with ECHO option

SET HELP FILE

The SET HELP FILE command specifies the location of the data file containing the help text for Inspect. This command is intended for system administrators wishing to offload files from $SYSTEM onto another volume.

```
SET HELP FILE [=] filename
```

filename

specifies the name of a file.

Default Value

By default, Inspect looks for a file named INSPHELP in the same subvolume as the Inspect object file. Typically, the Inspect object file is located in the SYSnn subvolume.

Related Command

SHOW with HELP FILE option

SET LOCATION FORMAT

The SET LOCATION FORMAT command controls how Inspect displays code locations. In addition, the SET LOCATION FORMAT command allows you to differentiate between statement numbers and labels within FORTRAN programs.

```
SET LOCATION FORMAT [ level ] [=] loc-fmt [, loc-fmt ]...


level:  one of

   BOTH   HIGH   LOW


loc-fmt:  one of

   INSTRUCTION[S]
   LINE[S] [ FILE [ ALL ] ] [ OFFSET ]
   STATEMENT[S] [ OFFSET ]
```

level

specifies the command-mode level whose location format you want to set. HIGH specifies high-level Inspect, LOW specifies low-level Inspect, and BOTH specifies both high-level and low-level Inspect. If you omit *level*, SET LOCATION FORMAT affects the current command mode.

loc-fmt [, *loc-fmt*]...

specifies one or more location formats for Inspect to use when displaying code locations.

INSTRUCTIONS

directs Inspect to display a code location as an octal offset from the base of its containing scope unit:

scope-path + %*octal-num*I

`LINE[S] [ FILE [ ALL ] ] [ OFFSET ]`

directs Inspect to display the scope path for a code location followed by the EDIT line number corresponding to the location:

scope-path.#*line-number*

The FILE clause directs Inspect to add the eight-character name of the source file; ALL directs Inspect to use the fully qualified file name:

scope-path.#*line-number* (*source-file*)

The OFFSET clause directs Inspect to add the octal instruction offset of a location from the start of its containing statement:

scope-path.#*line-number* + %*octal-num*I

The OFFSET clause is useful when execution has been suspended in the middle of a statement; this often happens with data breakpoints.

`STATEMENT[S] [ OFFSET ]`

directs Inspect to display the scope path for a code location followed by the location's statement number within its containing scope unit:

scope-path.*statement-number*

Additionally, the STATEMENT clause is used to determine if statement numbers or labels will be used for FORTRAN code locations. If the location format is STATEMENTS, then the location by statement-number is used. If the location format does not include STATEMENT, then statement-label is used.

For more information, see Section 11, Using Inspect With FORTRAN.

The OFFSET clause directs Inspect to add the octal instruction offset of a location from the start of its containing statement:

scope-path.*statement-number* + %*octal-num*I

Default Value

The default INSPLOCL file sets the location format as follows:

```
SET LOCATION FORMAT HIGH = LINES FILE OFFSET
SET LOCATION FORMAT LOW = INSTRUCTIONS
```

Usage Considerations

- When Inspect Uses the Location Format

  Inspect uses the location format whenever it displays a code location. Consequently, the location format affects the information provided by such commands as:

BREAK	ICODE	LIST BREAKPOINT
LIST PROGRAM	SOURCE	SET STATUS LINE25
SET STATUS SCROLL	TRACE	

- Code Locations in Scope Units without Symbols

  When Inspect displays a code location in a scope unit that does not have symbols, it cannot provide the scope path, line number, or statement number. For code locations without symbols, Inspect displays the name of the code block containing the location, followed by the octal instruction offset of the location from the block's base, regardless of the LOCATION FORMAT.

SET PRIV MODE

The SET PRIV MODE command enables you to perform operations requiring privileged system access. The SET PRIV MODE command is valid only if the user ID of the Inspect process is 255,255.

```
SET PRIV MODE [=] ON | OFF
```

`ON`

enables privileged debugging.

`OFF`

disables privileged debugging.

Usage Consideration

The SET PRIV MODE command provides an extra level of safety by requiring the user to be logged on as the super ID and having the SET PRIV MODE command set properly.

Related Command

SHOW with PRIV MODE option

SET PROMPT

The SET PROMPT command controls the format of the Inspect prompt.

```
SET PROMPT [=] [ prompt-item [ , prompt-item ]... ]


prompt-item:   one of

   string            ACCELERATOR STATE    COMMAND
   DIRECTORY         FN                   ICODE
   INSTRUCTION SET   LEVEL                PROCESSOR
   PROGRAM FILE      PROGRAM NAME         PROGRAM ORDINAL
   PROGRAM PID       RADIX                SOURCE
   STEP              SUBVOL[UME]          SYSTEM
   SYSTYPE           VOLUME


string:   one of

   " [ character ]... "
   ' [ character ]... '
```

prompt-item [, *prompt-item*]...

is a list of prompt items that defines the content of the Inspect prompt. When it issues a prompt, Inspect displays the prompt items in the order they appear in the list.

string

directs Inspect to display a string of text in the prompt. This string is a group of zero or more characters enclosed in either quotes (") or apostrophes ('). To include a quote in a quote-delimited string, use a pair of quotes. To include an apostrophe in an apostrophe-delimited string, use a pair of apostrophes.

ACCELERATOR STATE

directs Inspect to indicate the program state at the current location. This token applies only to accelerated programs. If the program is not running on a TNS/R or TNS/E system or has not been accelerated, nothing is shown. One of these accelerator state values is listed:

Register-exact
Memory-exact
Non-exact

COMMAND

directs Inspect to display the command number (as used by FC, XC, and LIST HISTORY).

`DIRECTORY`

directs Inspect to display the current OSS directory. Directory remains blank if there is no current OSS directory.

`FN`

directs Inspect to display “(FN)”. You can continue the previous low-level FN (Find Number) command by pressing RETURN.

`ICODE`

directs Inspect to display “(ICODE)”. You can continue the previous ICODE command by pressing RETURN.

`INSTRUCTION SET`

directs Inspect to display the name of the machine instructions currently being executed by your program.

One of: `TNS | TNS/R`

This indicates whether the program is executing TNS instructions or TNS/R instructions produced by the Axcel accelerator.

`LEVEL`

directs Inspect to display the command-mode level. For high-level mode, Inspect displays a dash; for low-level mode, it displays an underscore.

`PROCESSOR`

directs Inspect to display the machine family name and processor name on which the current program is executing.

Output is of the form: *`family (processor)`*

For example, `TNS/R (NSR-L)`.

`PROGRAM FILE`

directs Inspect to display the name of the current program’s program file.

`PROGRAM NAME`

directs Inspect to display the current program’s program name (as displayed by the LIST PROGRAM command).

`PROGRAM ORDINAL`

directs Inspect to display the current program’s program number (as displayed by the LIST PROGRAM command).

`PROGRAM PID`

directs Inspect to display the current program’s process ID in the form sys,cpu,pin.

`RADIX`

directs Inspect to display the current input radix: DEC, HEX, or OCTAL.

`SOURCE`

directs Inspect to display “(SOURCE)” if you can continue the previous SOURCE command by pressing RETURN.

`STEP`

directs Inspect to display “(STEP *count unit)*” if you can repeat the previous STEP command by pressing RETURN. Inspect displays the count if it is not one, and displays the unit if it is not the default for the current level (STATEMENTS is the high-level default; INSTRUCTIONS is the low-level default).

`SUBVOLUME`

directs Inspect to display the current default subvolume.

`SYSTEM`

directs Inspect to display the current default system.

`SYSTYPE`

directs Inspect to display the current systype, either Guardian or OSS.

`VOLUME`

directs Inspect to display the current default volume.

Default Value

The default INSPLOCL file sets the prompt as follows:

```
SET PROMPT = LEVEL,PROGRAM NAME,STEP,LEVEL
```

Usage Consideration

The maximum length of Inspect’s prompt is 70 characters. Any text exceeding 70 characters will be truncated.

Related Commands

- LIST PROGRAM on page 6-137
- SET STATUS LINE25 and SET STATUS SCROLL on page 6-187
- SHOW with PROMPT option on page 6-194

SET RADIX

The SET RADIX command changes the default radix (numeric base) for integer representations of input, output, or both. Inspect uses the default input radix to qualify integer values that are unqualified—that is, integers whose base is not specified. Each source language provides its own mechanism for specifying bases.

```
SET RADIX [ INPUT | OUTPUT ] [ level ] [=] radix

level:   one of

   BOTH   HIGH   LOW

radix:   one o

   DEC[IMAL]   HEX[ADECIMAL]   OCT[AL]
```

`INPUT`

specifies that radix applies only to input data.

`OUTPUT`

specifies that radix applies only to output data.

`level`

specifies the command-mode level whose radix you want to set. HIGH specifies high-level Inspect, LOW specifies low-level Inspect, and BOTH specifies both high-level and low-level Inspect. If you omit *`level`*, SET RADIX affects the current command mode.

`radix`

selects the new default radix, and is one of:

```
DECIMAL
HEXADECIMA
OCTAL
```

Default Values

- If you specify neither INPUT nor OUTPUT, Inspect sets both the input and output radixes to radix.
- If you omit level, SET RADIX affects the current command mode.
- The default INSPLOCL file sets the radixes as follows:

```
SET RADIX INPUT HIGH = DECIMAL
SET RADIX OUTPUT HIGH = DECIMAL
SET RADIX INPUT LOW = OCTAL
SET RADIX OUTPUT LOW = OCTAL
```

Usage Considerations

- Hexadecimal Numbers

  Inspect accepts the hexadecimal digits A through F in either uppercase or lowercase. To specify a hexadecimal number whose first digit is in the range A through F, you must prefix it with a zero or the hexadecimal qualifier (%h or %H). For example, once you've set your input radix to hexadecimal, the value 9FF is acceptable without qualification, but the value A00 must be entered as 0A00,%hA00, or %HA00.

- When Inspect Uses the Input Radix

  Inspect uses the current input radix to interpret values in expressions. It does not use the current input radix to interpret line numbers, statement numbers, breakpoint numbers, and other such measurement or identifier numbers.

Related Commands

- DISPLAY on page 6-33
- SET CHARACTER FORMAT on page 6-172
- SHOW with RADIX option on page 6-194

Example

This example demonstrates various SET RADIX commands using the integer variable GABRIEL. Note that the initial default radix is decimal:

```
-TALOBJ-DISPLAY gabriel
GABRIEL = 135
```

Switch both radixes and display:

```
-TALOBJ-SET RADIX = OCTAL; DISPLAY gabriel
GABRIEL = %207
```

Set GABRIEL to a binary value and display:

```
-TALOBJ-MODIFY gabriel = %b00110100; DISPLAY gabriel
GABRIEL = %64
```

Show GABRIEL in a new default radix:

```
-TALOBJ-SET RADIX = DECIMAL; DISPLAY gabriel
GABRIEL = 52
```

Modify it to show that the default input radix is 10:

```
-TALOBJ-MODIFY gabriel = 12; DISPLAY gabriel
GABRIEL = 12
```

Set the input radix to octal, and leave the output radix at decimal:

```
-TALOBJ-SET RADIX INPUT = OCTAL; DISPLAY gabriel
GABRIEL = 12
```

Enter a new octal value; displaying it results in decimal:

```
-TALOBJ-MODIFY gabriel = 45; DISPLAY gabriel
GABRIEL = 37
```

Enter a new value explicitly as decimal:

```
-TALOBJ-MODIFY gabriel= %D45; DISPLAY gabriel
GABRIEL = 45
```

SET SOURCE BACK and SET SOURCE FOR

The SET SOURCE BACK and SET SOURCE FOR commands control the defaults the SOURCE command uses when displaying source code. SET SOURCE BACK controls how much source code SOURCE displays preceding the requested location, and SET SOURCE FOR controls how much source code SOURCE displays.

```
SET SOURCE { BACK | FOR } [=] count


count:

   integer [ STATEMENT[S] | LINE[S] | INSTRUCTION[S] ]
```

count

specifies the amount of source code as a number of statements, source lines, or instructions.

Default Value

The default INSPLOCL file sets SOURCE BACK and SOURCE FOR as follows:

```
SET SOURCE BACK = 4 LINES
SET SOURCE FOR = 10 LINES
```

Related Commands

- SET SOURCERANGE on page 6-184
- SELECT SOURCE SYSTEM on page 6-169
- SHOW with SOURCE BACK option on page 6-194
- SHOW with SOURCE FOR option on page 6-194
- SOURCE on page 6-196

SET SOURCERANGE

The SET SOURCE RANGE command controls the default range used by the SOURCE SEARCH command.

```
SET SOURCE RANGE [=] range / range


range:   one of

   F   L   #line-number   statement-number
```

range / range

specifies the beginning and ending of the default range.

F

specifies the first line in the source file being searched.

L

specifies the last line in the source file being searched.

#line-number

specifies a given line in the source file being searched.

statement-number

specifies the line at which a given statement begins.

Default Value

The default INSPLOCL file sets SOURCE RANGE as follows:

```
SET SOURCE RANGE = F / L
```

Related Commands

- SELECT SOURCE SYSTEM on page 6-169
- SET SOURCE BACK and SET SOURCE FOR on page 6-183
- SHOW with SOURCE RANGE option on page 6-194
- SOURCE on page 6-196

SET SOURCE WRAP

The SET SOURCEþWRAP command controls whether the SOURCE command truncates or wraps long source lines by default. When source wrapping is on, the SOURCE command displays the full length of each source line, wrapping it onto the next screen line if necessary.

```
SET SOURCE WRAP [=] { ON | OFF }
```

Default Value

The default INSPLOCL file sets SOURCE WRAP as follows:

```
SET SOURCE WRAP = OFF
```

Related Commands

- SHOW on page 6-194
- SOURCE on page 6-196

SET STATUS ACTION

The SET STATUS ACTION command specifies actions that Inspect is to perform after it displays the event status message.

```
SET STATUS ACTION [ level ] [=] [ cmd-string ]


level:   one of

   BOTH   HIGH   LOW


cmd-string:   one of

   " command [ ; command ]... "
   ' command [ ; command ]... '
```

level

specifies the command-mode level whose status action you want to set. HIGH specifies high-level Inspect, LOW specifies low-level Inspect, and BOTH specifies both high-level and low-level Inspect. If you omit level, SET STATUS ACTION affects the current command mode.

cmd-string

specifies a string of Inspect commands to be performed. This command string is a group of one or more Inspect commands separated by semicolons (;) and enclosed in either quotes (") or apostrophes ('). To include a quote in a quote-delimited command string, use a pair of quotes. To include an apostrophe in an apostrophe-delimited command string, use a pair of apostrophes.

Usage Consideration

The SET STATUS ACTION command overrides the SOURCE ON command.

Default Values

- If you omit *level*, SET STATUS ACTION affects the current command mode.
- If you omit *cmd-string*, Inspect sets the status action to nothing.
- The default INSPLOCL file sets the status action to nothing, as follows:

```
SET STATUS ACTION BOTH
```

Related Commands

- SET STATUS LINE25 and SET STATUS SCROLL on page 6-187
- SHOW with STATUS ACTION option on page 6-194

SET STATUS LINE25 and SET STATUS SCROLL

The SET STATUS LINE25 and SET STATUS SCROLL commands control the information Inspect displays in the event-status message. The SET STATUS LINE25 command controls what information appears on the 25th line of the terminal; the SET STATUS SCROLL command controls what information appears in the scrolling portion of the terminal.

```
SET STATUS { LINE25 | SCROLL } [ level ] [ OUT filename ] [=]
   [ status-item [ , status-item ]... ]


level:   one of

   BOTH   HIGH   LOW


status-item:   one of

   string            ACCELERATOR STATE  EVENT
   INSTRUCTION SET   LANGUAGE           LOCATION
   NEW LINE          PROCESSOR          PROGRAM FILE
   PROGRAM NAME      PROGRAM ORDINAL    PROGRAM PID
   SCOPE             STATE              SYSTYPE
   TYPE


string:   one of

   " [ character ]... "
   ' [ character ]... '
```

level

specifies the command mode level whose status message you want to set. HIGH specifies high-level Inspect, LOW specifies low-level Inspect, and BOTH specifies both high-level and low-level Inspect. If you omit level, SET STATUS LINE25 and SET STATUS SCROLL affect the current command mode.

`status-item [ , status-item ]...`

is a list of status items that defines the lineþ25 or scrolling content of the Inspect event status message. Inspect displays the status items in the order that they appear in the list.

`OUT filename`

allows you to specify what out file you want data to go to when an event occurs.

`string`

directs Inspect to display a string of text. This string is a group of zero or more characters enclosed in either quotes (") or apostrophes ('). To include a quote in a quote-delimited string, use a pair of quotes. To include an apostrophe in an apostrophe-delimited string, use a pair of apostrophes.

`ACCELERATOR STATE`

directs Inspect to indicate the program state at the current location. This token applies only to accelerated programs running on a TNS/R system. If the program is not running on a TNS/R system or has not been accelerated, nothing is shown. One of these values is listed:

Register-exact
Memory-exact
Non-exact

`EVENT`

directs Inspect to display the type of event that occurred:

```
ABEND
BREAKPOINT #brkpt-number: code-location
CALL
MEMORY ACCESS BREAKPOINT
brkpt-number: data-location [OCCURRED AT code-location]

PATHWAY ERROR CODE: error-code
STEP [ IN | OUT ]
STOP
TRAP trap-number - trap-text
```

`INSTRUCTION SET`

directs Inspect to display the type of machine instructions currently being executed by your program.

One of: `TNS | TNS/R`

This indicates whether the current program is executing TNS instructions or TNS/R instructions.

`LANGUAGE`

directs Inspect to display the current language.

`LOCATION`

directs Inspect to display the location at which the event occurred. Inspect uses the format specified by SET LOCATION FORMAT to display the location.

`NEW LINE`

directs Inspect to go to the next screen line. Inspect ignores NEW LINE for SET STATUS LINE25.

`PROCESSOR`

directs Inspect to display the machine family name and processor name the current program is executing on.

Output is of the form: *`family (processor)`*

`family` is one of:

```
TNS
TNS/R
```

`processor` is one of:

```
TNSII
TXP
VLX
CLX
Cyclone
NSR-L
NSR-N
NSE-P
```

For example: `TNS/R (NSR-L)` and `TNS/E (NSE-P)`

`PROGRAM FILE`

status message; directs Inspect to display the name of the current program's file.

`PROGRAM NAME`

directs Inspect to display the current program's process name. If the program is not named, Inspect used the name of the current program's program file.

`PROGRAM ORDINAL`

directs Inspect to display the current program's program number (as displayed by the LIST PROGRAM command).

`PROGRAM PID`

directs Inspect to display the current program’s process ID in the form *sys,cpu,pin.*

`SCOPE`

directs Inspect to display the current scope path.

`STATE`

directs Inspect to display the current program's execution state: RUN, HOLD, STOP or GONE.

For more information about execution states, see Section 2, Inspect Concepts.

`SYSTYPE`

directs Inspect to display the current systype, either Guardian or OSS.

`TYPE`

directs Inspect to display the current program’s type.

For more information, see Scope Paths on page 2-12.

Default Values

- If you omit level, SET STATUS LINE25 and SET STATUS SCROLL affect the current command mode.
- If you do not specify a list of status items, Inspect sets the specified status message (line 25 or scrolling) to nothing.
- The default INSPLOCL file sets the scrolling and line 25 portions of the event-status message as follows (note that it sets the line 25 portion to display nothing):

```
SET STATUS SCROLL BOTH = "INSPECT  ",EVENT,NEW LINE,PROGRAM PID,"  ", &
                         PROGRAM NAME,"  ",LOCATION
SET STATUS LINE25 BOTH
```

Usage Considerations for SET STATUS LINE25

- When Inspect Updates Line 25

  Inspect updates the line 25 status message when a debug event occurs. In addition, Inspect updates the line 25 status message when the value of one of its status items changes. For example, if you set the line 25 status message to include the current language, Inspect will update the line 25 status message whenever the current language changes.

- Maximum Length of the Line 25 Status Message

  The line 25 status message accommodates up to 64 characters. If the list of status items you specify results in a longer status message, Inspect truncates it to 64 characters.

- Maximum Amount Written in the Event-Status Message

  The maximum amount that is written out in the event-status message is determined by the record length of the device being written to.

Related Commands

- LIST PROGRAM on page 6-137
- SET PROMPT on page 6-178
- SHOW with STATUS LINE25 option on page 6-194
- SHOW with STATUS SCROLL option on page 6-194

SET SUBPROC SCOPING

The SET SUBPROC SCOPING command allows you to specify whether Inspect first looks in a subprocedure for an identifier or first looks in the encompassing procedure.

```
SET SUBPROC SCOPING [=] [ SUBLOCAL | LOCAL ]
```

`SUBLOCAL`

specifies that the sublocal identifier will take precedence.

`LOCAL`

specifies that the local identifier will take precedence.

Usage Considerations

- If SUBPROC SCOPING is set to SUBLOCAL, when a sublocal in the current TAL subprocedure has the same name as a local in the containing procedure, the sublocal will take precedence. The local cannot be accessed.
- If SUBPROC SCOPING is set to LOCAL, when a sublocal in the current TAL subprocedure has the same name as a local in the containing procedure, the local will take precedence. The sublocal can still be accessed by qualifying it with the subprocedure name.

Related Commands

- DISPLAY on page 6-33
- SHOW with SUBPROC SCOPING option on page 6-194

Example

This example illustrates the SET SUBPROC SCOPING command. Note that the current location is within subprocedure “s”.

```
-TEST(STEP)-SOURCE #0 FOR 24 LINES
   #1           ?PAGE "PROC p"
   #2           PROC p MAIN;
   #3
   #4           BEGIN
   #5
   #6           INT
   #7              i := 42;
   #8
   #9           ?PAGE "SUBPROC s of PROC p"
   #10           SUBPROC s;
   #11
   #12          BEGIN
   #13
   #14          INT
   #15             i := 101;
   #16
  *#17          END;-- of SUBPROC s
   #18          -- ********************End of SUBPROC s *********************
   #19          ?PAGE "PROC p"
   #20
   #21             CALL s;
   #22
   #23          END;-- of PROC p
   #24          -- ******************** End of PROC p **********************
-TEST-SHOW SUBPROC SCOPING
SUBLOCAL
-TEST-DISPLAY i
I = 101
-TEST-SET SUBPROC SCOPING = LOCAL
-TEST-DISPLAY i
** Inspect warning 99 ** Access is local (sublocal reference must be
qualified): I
I = 42
-TEST-DISPLAY S.I
S.I = 101
```

SET SYSTYPE

The SET SYSTYPE command allows you to change the current systype of Inspect.

```
SET SYSTYPE { GUARDIAN | OSS }
```

Usage Consideration

Inspect may override any user selected systype when a subsequent event is received.

SET TRACE

The SET TRACE command controls the dynamic trace facility of Inspect. When dynamic tracing is activated, Inspect displays trace information concerning the current program as it executes. SET TRACE enables you to activate this facility and control the level of tracing.

Note. The SET TRACE command is invalid for Pathway requester programs.

```
SET TRACE trace-level [=] { ON | OFF }


trace-level:   one of

   ARGUMENT[S]   SCOPE[S]   STATEMENT[S]
```

trace-level

specifies the level of dynamic tracing you want to set.

ARGUMENT

specifies dynamic tracing of arguments. When tracing of arguments is on, Inspect displays a trace message each time execution control passes to a scope unit as the result of a call to the scope unit. The trace message displays the arguments (if any) specified in the call. In general, the ARGUMENT option is used with the SCOPE option, however, SET TRACE ARGUMENTS is inactive when SET TRACE SCOPES is off. SET TRACE ARGUMENT is resource-intensive; it should be used over a limited execution range in your program.

SCOPE

specifies dynamic tracing of scope units. When scope tracing is on, Inspect displays a trace message each time execution control passes from one scope unit to another.

STATEMENT

specifies dynamic tracing of statements. When statement tracing is on, Inspect displays a trace message each time a statement executes. Inspect uses the format specified by SET LOCATION FORMAT to display the location of the statement.

Default Value

The default INSPLOCL file sets the dynamic trace controls as follows:

```
SET TRACE ARGUMENTS = OFF
SET TRACE SCOPES = OFF
SET TRACE STATEMENTS = OFF
```

Usage Considerations

- The SET TRACE command is ignored when the STEP command is used.
- SCOPE TRACE and ARGUMENT TRACE information is not reported when a breakpoint is hit.

Related Commands

- RESUME on page 6-158
- SHOW with TRACE ARGUMENT option on page 6-194
- SHOW with TRACE SCOPE option on page 6-194
- SHOW with TRACE STATEMENTS option on page 6-194

SHOW

The SHOW command displays the status of one or all the Inspect parameters controlled by the SET command.

```
SHOW { ALL [ AS COMMAND[S] ] }
     { set-object            }


set-object:   one of

   CHARACTER FORMAT
   DEREFERENCE DEPTH
   ECHO { ALL | ALIAS[ES] | HISTORY | KEYS }
   HELP FILE
   LOCATION FORMAT [ level ]
   PRIV MODE
   PROMPT
   RADIX [ INPUT | OUTPUT ] [ level ]
   SOURCE { ALL | BACK | FOR | RANGE | WRAP }
   STATUS { ALL | ACTION | LINE25 | SCROLL } [ level ]
   SUBPROC SCOPING
   TRACE { ALL | ARGUMENT[S] | SCOPE[S] | STATEMENT[S] }


level:   one of

   BOTH   HIGH   LOW
```

Usage Consideration

Using AS COMMANDS

The AS COMMANDS clause directs SHOW to display settings as executable Inspect commands. If you use the AS COMMANDS clause with the OUT command, you can create a command file. You can then incorporate this file in an OBEY file or the INSPLOCL or INSPCSTM customization files; for example:

```
-PRG-SHOW /OUT mysets/ ALL AS COMMANDS
```

Related Commands

- ENV on page 6-81
- LIST on page 6-129
- SET on page 6-171

SIGNALS

The SIGNALS command displays signal information for the current program; it does not support Guardian processes. The SIGNALS command is an alias for the INFO SIGNALS command.

```
SIGNAL[S] [ * | signal-id [, signal-id...] ]
                             [ [, ] DETAIL ]
```

`*`

requests the status of all signals of the current program.

`signal-id`

requests the status of a specific signal, one of:

SIGABRT	SIGALRM	SIGFPE
SIGHUP	SIGILL	SIGINT
SIGIO	SIGKILL	SIGPIPE
SIGQUIT	SIGRECV	SIGSEGV
SIGTERM	SIGUSR1	SIGUSR2
SIGCHLD	SIGCONT	SIGSTOP
SIGTSTP	SIGTTIN	SIGTTOU
SIGABEND	SIGLIMIT	SIGSTK
SIGMEMMGR	SIGNOMEM	SIGMEMERR
SIGTIMEOUT		

Default Value

If you enter SIGNALS alone, Inspect displays information on all the signals of the current program.

SOURCE

The SOURCE command's primary function is to display source text. It performs other functions, including searching for source text, displaying text from an arbitrary file, and allowing you to redefine the location of source files.

```
SOURCE [ source-locator ] [ limit-spec ]...
   [ file-locator ] [ WRAP ]


source-locator:   one of

   AT code-location
   ICODE [ AT code-location ]
   [ LINE ] #line-number
   [ STATEMENT ] statement-number
   SEARCH string [ CASE ] [ position / position ]


limit-spec:   one of

   FOR count [ STATEMENT[S] | LINE[S] | INSTRUCTION[S] ]
   BACK count [ STATEMENT[S] | LINE[S] | INSTRUCTION[S] ]
   / position


position:   one of

   F   L   #line-number   statement-number


string:   one of

   " [ character ]... "
   ' [ character ]... '


file-locator:   one of

   FILE file-name
   LOCATION code-location
   SCOPE scope-number
```

source-locator

specifies what source text to display or search for.

AT *code-location*

specifies the source text corresponding to the given code location.

`ICODE [ AT` *`code-location`* `]`

specifies instruction-code presentation of the source text corresponding to the code location given in the AT clause.

`LINE` *`#line-number`*

specifies the source text corresponding to the given line number. Inspect uses the current scope path to qualify this line number unless you specify *`file-locator`*.

`[ STATEMENT ]` *`statement-number`*

specifies the source text corresponding to the given statement number. Inspect uses the current scope path to qualify this statement number unless you specify *`file-locator`*.

`limit-spec`

specifies the amount of source text to display.

You cannot specify a limit specifier when using SOURCE SEARCH.

`FOR` *`count`* `[ STATEMENT[S] | LINE[S] ]`

specifies the amount of source text to display. The *`count`* parameter specifies the number of units (statements or lines) to display. Inspect selects the default unit based on the current setting of the LOCATION FORMAT environment parameter. If the setting specifies STATEMENTS, Inspect uses STATEMENTS as the default unit; otherwise, it uses LINES.

`BACK` *`count`* `[ STATEMENT[S] | LINE[S] ]`

specifies the amount of source text preceding source-locator to display. The *`count`* parameter specifies the number of units (statements or lines) to display. If you omit the unit, Inspect selects a default based on the current setting of the LOCATION FORMAT environment parameter. If the setting specifies STATEMENTS, Inspect uses STATEMENTS as the default unit; otherwise, it uses LINES.

You cannot specify the BACK clause when using SOURCE ICODE.

`/ position`

specifies one endpoint of a range whose other endpoint is *`source-locator`*. Inspect displays the source text in this range. You can specify the endpoint by line number, statement number, or by the letters F or L. F specifies the first line in the file; L specifies the last line in the file.

You cannot specify this clause with the FOR or BACK clauses.

When you use this clause in a SOURCE ICODE command, you cannot use the letters F or L to specify the endpoint of the range.

file-locator

directs Inspect to read from a file other than the one containing the source text for the current scope path. In this context, the SOURCE command is used to browse through edit files. Inspect does not annotate the source with information such as statement numbers, memory-exact points or the current location. The *file-locator* clause accepts both Guardian filenames and OSS pathnames, depending on the current systype of Inspect. If no *file-locator* is specified, Inspect uses the source file specified in the object file for the current scope, which can be either a Guardian file name or an OSS pathname.

You cannot specify a file locator when using SOURCE AT or SOURCE ICODE.

FILE *file-name*

directs Inspect to read from the given file.

You cannot specify the FILE clause when using SOURCE STATEMENT.

LOCATION *code-location*

directs Inspect to read from the file containing the source text for the given code location.

You cannot specify the LOCATION clause when using SOURCE STATEMENT.

SCOPE *scope-number*

directs Inspect to read from the file containing the source text for the active scope unit whose scope number (as shown by the TRACE command) is scope-number.

WRAP

directs Inspect to display the full length of each source line, wrapping it onto the next line if necessary.

Default Values

- If you do not specify *source-locator*, the SOURCE command uses the current scope path to determine what source text to display:
  - If the current scope path denotes an active scope unit, the SOURCE command displays source text associated with the next code location in the scope unit.
  - If the current scope path denotes an inactive scope unit, the SOURCE command displays source text associated with the first code location in the scope unit.
- If you do not specify *limit-spec*, the SOURCE command uses the current values of the SOURCE FOR and SOURCE BACK environment parameters. You can use the SET command to change these parameters.

- If you do not specify *`file-locator`*, the SOURCE command reads from the file associated with *`source-locator`*.
- If you do not specify WRAP, the SOURCE command uses the current value of the SOURCE WRAP environment parameter. You can use the SET command to change this parameter.

Usage Considerations

- Components of the Source Display

  When you use the SOURCE command, Inspect displays the source text corresponding to the location you provide. In addition, Inspect also displays:

  - An asterisk next to the source line containing the current location.
  - Breakpoint numbers next to source lines containing break locations.
  - Statement numbers if the LOCATION FORMAT session parameter includes STATEMENTS.
  - Line numbers if the LOCATION FORMAT session parameter includes LINES.

- Continuing the Last SOURCE Command

  After you enter a SOURCE command, you can continue it by pressing RETURN at the next Inspect prompt. The ability to repeat continues until you enter any other Inspect command.

  When you repeat a SOURCE command, Inspect continues from the source line where the previous SOURCE command ended.

  When you repeat a SOURCE SEARCH command, Inspect searches for the next occurrence of the original string.

- Accessing Source Text

  In order for SOURCE to access and display the source text, the scope unit containing source-locator must have symbol information associated with it. If no symbol information is available, Inspect cannot display any source text and therefore issues the error message:

```
** Inspect error 36 ** No symbols available in scope:  scope-unit
```

- Timestamps of Source Files

  When Inspect first opens a source file, it checks the file's current modification timestamp. If this timestamp differs from the timestamp recorded in the program file, Inspect issues this warning:

```
** Inspect warning 49 ** Timestamp mismatch for file-name
   Source modification time at present:     current-timestamp
   Source modification time at compilation: compile-timestamp
```

Inspect issues this warning once when it first opens the file; subsequent accesses to the file do not produce the warning.

Modifying a source file after it was compiled will result in a timestamp mismatch. In this case, the correspondence between code locations and source lines might be invalid.

Moving or renaming a source file and then using ADD SOURCE ASSIGN to associate the new name with the object code can also result in a timestamp mismatch, even when the source itself has not changed. To avoid a mismatch of this type, use either the SAVEALL or SOURCEDATE clause of the File Utility Program (FUP) DUPLICATE command. For more information, see the *File Utility Program (FUP) Reference Manual*.

- Types of Source Files

  Source files must be EDIT files. Inspect reports an error if a source file is not of this type.

- Renumbering a Source File

  If you renumber the lines in a source file after compiling it, the correspondence between line numbers and code locations might become invalid, therefore causing the SOURCE command to produce incorrect results.

- Pressing the BREAK Key

  You can halt a source display at any time by pressing the BREAK key. Inspect stops displaying source text and issues an Inspect prompt.

- Displaying Source from Multiple Files

  A single SOURCE command can display source text from one file only. Inspect does not interpret toggles, source directives, or copy directives in the source file. If Inspect encounters an end-of-file or beginning-of-file when displaying the source text, it shortens the display accordingly.

- Displaying Source with the FILE Clause

  The intent of the SOURCE FILE form of the SOURCE command is to allow you to browse through EDIT files while in Inspect. This form of the SOURCE command does not use symbol information; hence, it does not annotate the listing when the file happens to be a source file for the program.

Usage Consideration for Accelerated Programs on TNS/R Systems

When debugging accelerated programs, the SOURCE command annotates the listed statements to mark statements that are register-exact points with a “@” and those that have been “Deleted” (that is, are not memory-exact points) with a “–”.

Note. The annotation character is listed in the same column that Inspect lists the asterisk when marking the current location. The asterisk always takes precedence; information about the program state at the current location is available from the ACCELERATOR status/prompt token and the LIST PROGRAM DETAIL and the INFO LOCATION commands.

Usage Consideration for Accelerated Programs on TNS/E Systems

The SOURCE command shows register-exact points(“@”) and deleted statement annotations (“–”) for an OCA process, but not for an OCA snapshot.

Related Commands

- ADD SOURCE ASSIGN on page 6-14
- DELETE SOURCE ASSIGN on page 6-31
- DELETE SOURCE OPEN on page 6-32
- LIST SOURCE ASSIGN on page 6-141
- LIST SOURCE OPEN on page 6-142
- SELECT SOURCE SYSTEM on page 6-169
- SET LOCATION FORMAT on page 6-175
- SET with SOURCE option on page 6-171
- SOURCE ASSIGN on page 6-202
- SOURCE OPEN on page 6-208
- SOURCE SYSTEM on page 6-211

Examples

1. When the LOCATION FORMAT is set to STATEMENTS, the SOURCE command does not list statement numbers for statements deleted by TNS optimizations. In this example, the statement number 15 is not listed because it was deleted by optimizations.

```
12                 z := x;
13                 z := y;
14                 GOTO bar;
                           GOTO foo;
16                  z := 1;
```

2. For an accelerated program on TNS/R systems, when the LOCATION FORMAT is set to LINES, the SOURCE command uses a dash (-) to annotate lines deleted by accelerator optimizations. In this example, the line number 80 has a dash because it was deleted by optimizations.

```
@#76               a :=1;
 #77               CALL s1;
@#78               CALL s2 ( a, a, a );
@#79               CALL s3;
-#80               b :=a;
 #81
 #82                CALL PROC1 (1, 2D, sptr );
```

3. This example illustrates that when a breakpoint at STOP or ABEND is recognized, Inspect shows the current location at the call statement. The annotation character, “*”, denotes the current location. In this example, assume a breakpoint at ABEND has been hit and the user typed “SOURCE.”

```
100 IF j = 17
200  THEN
300    BEGIN
500      IF e <> s OR j <> e
600      THEN
*700        CALL ABEND;
800    END
900  ELSE
910    BEGIN
920      J := 17;
```

SOURCE ASSIGN

The SOURCE ASSIGN command redefines the location of source files and sets or displays source assignments from the source-assignment list for the current Inspect session. The SOURCE ASSIGN command is a synonym for the ADD SOURCE ASSIGN and LIST SOURCE ASSIGN commands. For more information, see ADD SOURCE ASSIGN and LIST SOURCE ASSIGN commands.

When a program is compiled, the fully qualified names of the source files that compose it are recorded as part of the symbol information. Inspect uses this information to determine what file to retrieve source text from. If a source file has been moved since a program was compiled, Inspect will be unable to locate source text to display. The SOURCE ASSIGN command enables you to inform Inspect where to find source files when their location has changed.

```
SOURCE ASSIGN[S]  [  [  original-name,  ]  new-name  ]


original-name:  one of

   [ \system. ] $volume [ .subvolume [ .file ] ]
   [ \system. ] $process [ .#qual-1 [ .qual-2 ] ]
   [ \system. ] cpu, pin
   [ \system. ] $volume.#number
   /oss-pathname [/oss-pathname ...]


new-name:

   [\system.] $volume [ .subvolume [ .file ]
   /oss-pathname [/oss-pathname ...]
```

original-name

specifies the name of a volume, subvolume, file (permanent or temporary), or process where Inspect would normally look for source code. Note that the volume name is required for a permanent or temporary file.

new-name

specifies the name of the volume, subvolume, or file where you want Inspect to look for source code when it would normally look in *original-name*. Note that the volume is required.

new-name must be qualified down to the same level as *original-name*. That is, if *original-name* is a volume,*new-name* must be a volume; if *original-name* is a subvolume, *new-name* must be a subvolume; if *original-name* is a file or process, *new-name* must be a file.

Default Values

- If you do not specify *original-name* or *new-name*, SOURCE ASSIGN displays the source assignments from the source-assignment list for the current Inspect session.
- If you do not specify *original-name*, Inspect uses the same source file with the current scope.

Related Commands

- ADD SOURCE ASSIGN on page 6-14
- DELETE SOURCE ASSIGN on page 6-31
- LIST SOURCE ASSIGN on page 6-141
- SELECT SOURCE SYSTEM on page 6-169
- SOURCE on page 6-196
- SOURCE SYSTEM on page 6-211

SOURCE ICODE

The SOURCE ICODE command lists the instruction mnemonics corresponding to listed source text.

```
SOURCE ICODE [ AT code-location ] [ limit-spec ]  [ WRAP ]


limit-spec:  one of

   FOR count [ STATEMENT[S] | LINE[S] ]
   BACK count [ STATEMENT[S] | LINE[S] ]
```

`AT` *`code-location`*

specifies a code location relative to which source and instructions are to be listed.

`limit-spec`

specifies the amount of source text to display.

`FOR` *`count`* `[ STATEMENTS | LINES ]`

specifies the amount of source text to display. The *count* parameter specifies the number of units (statements or lines) to display. Inspect selects the default unit based on the current setting of the LOCATION FORMAT environment parameter. If the setting specifies STATEMENTS, Inspect uses STATEMENTS as the default unit; otherwise, it uses LINES.

`BACK` *`count`* `[ STATEMENTS | LINES ]`

specifies the amount of source text preceding *source-locator* to display. The *count* parameter specifies the number of units (statements or lines) to display. If you omit the unit, Inspect selects a default based on the current setting of the LOCATION FORMAT environment parameter.

`WRAP`

directs Inspect to display the full length of each source line, wrapping it onto the next line if necessary.

Default Value

The SOURCE ICODE command lists the same range of lines as listed by the SOURCE command.

Usage Considerations

- When displaying source and instruction mnemonics, Inspect lists source lines until the beginning of a statement/verb is encountered, at which point instructions for the preceding statement/verb are listed.
- Lines that do not generate code, such as comments, may therefore be listed before the instructions for the preceding statement.
- Changing the current scope to an active procedure on the stack changes the default display location.
- The SOURCE ICODE command requires that a program be compiled with symbols. Use the ICODE command if your program does not have symbols.

Usage Considerations for TNS/R Programs

- When debugging accelerated programs, the SOURCE ICODE command marks TNS instructions which are at memory-exact points and register exact-points. These symbols are used:

>	memory-exact point
@	register-exact points

- When debugging accelerated programs, the SOURCE ICODE command annotates the listed statements to mark statements that are register-exact points and those that have been “Deleted” (that is, are not memory-exact points). The annotation character is listed in the column before the line/statement number:

-	The statement is deleted (that is, it is not a memory-exact point).
@	The statement is a register-exact point; the RESUME AT command and register modification commands can be used as such statements.

- When debugging TNS/R native programs, the SOURCE ICODE command displays a “-” character next to RISC instructions which are from previous source lines and a “+” next to RISC instructions which are from subsequent lines. Lines containing RISC instructions also contain the source file line number that the instruction is for.

Related Command

ICODE

Example

1. This example illustrates the SOURCE ICODE command.

```
-PROGRAM-SOURCE ICODE AT #open^file FOR 7 statements
#398      INT PROC   open^file( fcb, fname^int );
  ADDS  +013         LADR  L+020         LLS     01
  PUSH   700         ADDS   +025
#399      STRING   .EXT fcb;
#400      INT          .fname^int;
#401    BEGIN
#402      STRING        buf[ 0:10 ];
#403      INT           dtype;
#404      INT           error;
#405      INT           error^subcode;
#406      INT           file^code;
#407      INT           fn;
#408      STRING       .fname^ext[ 0:EXT^FNAME^SIZE ];
#409      INT           fname^len;
#410      INT           reclen;
#411      INT           version;
#412
#413      fname^len := FNAMECOLLAPSE( fname^int, fname^ext );
  LADR  L-003,I       LADR  L+014,I       PUSH    711
  XCAL    121         STOR  L+015
#414        CALL DEVICEINFO( fname^int, dtype, reclen );
  LADR  L-003,I       LADR  L+007         LADR  L+016
  PUSH    722         XCAL    043
#415        IF dtype.DEVTYPE^TYPE <> DEVTYPE^DISC
#416          THEN
  LOAD  L+007         LRS     06          ANRI   +077
  CMPI   +003         BEQL   +010
#417          CALL ER^Write( ERR^NOT^DISC^FILE, 0, ,
 #417.1                          fname^ext, fname^len )
  LDI    +101         ZERD                LADR  L+014,I
  LOAD  L+015         ZERD                LDI    +154
  PUSH    777         XCAL    057
#418        fn := -1;  LDI    -001         STOR  L+013
#419          CALL OPEN( fname^int, fn, OPEN^READONLY );
  LADR  L-003,I       LADR  L+013         LDLI   +004
  PUSH    722         ADDS   +006         LDLI   +340
  LDI    -011         PUSH    711         XCAL    222
```

2. This example illustrates the SOURCE ICODE command listing TNS/R native programs.

```
-PROGRAM-SOURCE ICODE FOR 3 STATEMENTS
*#10.000   Proc P( u, v, w, x, y);

    10.000     addiu   $sp,$sp, -32
    10.000     sw      $4,32($sp)
    10.000     sw      $5,36($sp)

  #11.000     int(32)  u;
  #12.000     int(32)  v;
  #13.000     int(32)  w;
  #14.000     int(32)  x;
  #15.000     int(32)  y;
  #16.000   Begin
  #17.000     m := u + v + w + x;

     17.000     lw     $15,35($sp)
     17.000     lw     $14,32($sp)
   - 10.000     sw     $6, 40($sp)
     17.000     lw     $25,40($sp)
   - 10.000     sw     $7, 44($sp)
     17.000     lw     $9, 44($sp)
     17.000     add    $24,$14,$15
   + 18.000     lw     $11,48($sp)
     17.000     add    $8,$24,$25
   - 10.000     sw     $31,28($sp)
     17.000     add    $10,$8,$9
     17.000     sw     $10,16($gp)

 #18.000     call p(u, v, w, x, y);

     18.000     move     $5,15
     18.000     move     $4,14
     18.000     move     $6,25
     18.000     move     $7,9
     18.000     jal      0x70000290
     18.000     sw       $11,16

-PROGRAM-
```

SOURCE OFF

The SOURCE OFF command disables automatic source display at each event. SOURCE OFF is equivalent to the alias SOURCEOFF and the command SET STATUS ACTION HIGH.

```
SOURCE OFF
```

Related Commands

- SET STATUS ACTION on page 6-186
- SOURCE ON on page 6-208

SOURCE ON

The SOURCE ON command enables automatic source display at each event. SOURCE ON is equivalent to the command SET STATUS ACTION HIGH = "SOURCE FOR 1".

```
SOURCE ON
```

Related Commands

- SET STATUS ACTION on page 6-186
- SOURCE OFF on page 6-207

SOURCE OPEN

The SOURCE OPEN command displays the names of the files currently open as a result of previous SOURCE commands.

The SOURCE OPEN command is a synonym for the LIST SOURCE OPEN command.

```
SOURCE OPEN[S]
```

Related Commands

- DELETE SOURCE OPEN on page 6-32
- LIST SOURCE OPEN on page 6-142
- SOURCE on page 6-196

SOURCE SEARCH

The SOURCE SEARCH command displays source text that matches a specified string.

```
SOURCE SEARCH string [ CASE ] [ position / position ]
    [ file-locator ] [ WRAP ]
```

`SEARCH` *`string`* `[ CASE ] [` *`position / position`* `]`

specifies the source text corresponding to the first match of a given string. Inspect searches for the string in the source file containing the current scope path unless you specify file-locator.

string

is a string of characters enclosed in quotes. To include a quote in the string, use a pair of quotes.

CASE

directs Inspect to distinguish between uppercase and lowercase letters as it searches. If you omit CASE, Inspect does not differentiate uppercase from lowercase.

position / position

specifies the range of lines to search. The first position denotes the start of the range, and the second position denotes the end of the range. You can specify a position by line number, statement number, or by the letters F or L. The letter F specifies the first line in the file; L specifies the last line in the file.

If the ending position precedes the starting position, Inspect searches backwards through the range.

If you do not specify a range, Inspect uses the current value of the SOURCE RANGE environment parameter. You can use the SET command to change this parameter.

file-locator

directs Inspect to read from a file other than the one containing the source text for the current scope path. An OSS pathname is valid as a file-locator if the current systype is OSS.

WRAP

directs Inspect to display the full length of each source line, wrapping it onto the next line if necessary.

Usage Consideration

To find the next occurrence of a string, press the return key as illustrated.

```
-OBJECT-SOURCE SEARCH 'owner_id'
#20             int owner_id = 0;
-OBJECT-
#29             owner_id = (int) val3 >> 16;
-OBJECT-
#30             val2 = proc2(val1, owner_id);
```

Examples

This examples for the SOURCE SEARCH command are based on the source code. LOCATION FORMAT has been set to LINES, STATEMENTS to display both edit line and statement numbers.

```
-PROGRAM-SOURCE FOR 5 STATEMENTS
  *3    #25             val3 = 0x7DF4;
   4    #26             val1 = 0;
        #27
        #28
   5    #29             owner_id = (int) val3 >> 16;
   6    #30             val2 = proc2(val1, owner_id);
        #31
        #32             /* test should match first compare. */
   7    #33             if ( val2 == 0 )
```

1. Here are two examples of using SOURCE SEARCH to find a string of text within a source file.

```
-PROGRAM-SOURCE SEARCH "owner_id"
#20            int owner_id = 0;
-PROGRAM-SOURCE SEARCH 'owner_id'
#20             int owner_id = 0;
```

2. Here are two examples of SOURCE SEARCH which show the effect of the case clause. When CASE is used, note that the search is restricted. When a string is not located, Warning 203, “String not found,” will be issued.

```
-OBJECT-SOURCE SEARCH 'OWNER_ID'
#PROGRAM             int owner_id = 0;
-OBJECT-COMMENT OWNER_ID does not appear in upper-case anywhere
-OBJECT-COMMENT in the source. Warning 203 will be given.
-OBJECT-SOURCE SEARCH 'OWNER_ID' CASE
 ** Inspect warning 203 ** String not found
```

3. These three examples illustrate the use of the SOURCE SEARCH command when specifying position.

```
-PROGRAM---COMMENT Using the letters F and L.
-PROGRAM-SOURCE SEARCH 'owner_id' F/L
#20             int owner_id = 0;
-PROGRAM---COMMENT Using an edit line number and the letter L.
-PROGRAM-SOURCE SEARCH 'owner_id' #21/L
#29             owner_id = (int) val3 >> 16;
-PROGRAM---COMMENT Using a statement number and an edit line number.
-PROGRAM-SOURCE SEARCH 'owner_id' 3/#33
#29             owner_id = (int) val3 >> 16;
```

4. This example illustrates backward searching.

```
-PROGRAM-SOURCE SEARCH 'owner_id' #34/#20
#30             val2 = proc2(val1, owner_id);
-PROGRAM-
#29             owner_id = (int) val3 >> 16;
-PROGRAM-
#20             int owner_id = 0;
-PROGRAM-SOURCE SEARCH 'owner_id' #34/F
#30             val2 = proc2(val1, owner_id);
-PROGRAM-
#29             owner_id = (int) val3 >> 16;
-PROGRAM-
#20             int owner_id = 0;
```

5. This example illustrates the use of FILE as the file-locator to search for source from an arbitrary file.

```
-PROGRAM-SOURCE SEARCH 'readupdate' F/L FILE $system.system.extdecs
#4684       ?SECTION READUPDATE
```

6. This example shows using the SOURCE SEARCH command with SCOPE as the file-locator.

```
-PROGRAM-TRACE
Num Lang Location
  0  C   #proc2.1, #proc2.#61
  1  C   #main.6, #main.#30
  2  C   #_MAIN.3, #_MAIN.#64.001
-PROGRAM-SOURCE SEARCH 'owner_id' 1/7 SCOPE 1
#20             int owner_id = 0;
-PROGRAM-SOURCE
        #57
        #58         int proc2 (x,y)
        #59             int x;
        #60             long y;
  *1    #61         {
   2    #62          x = x + (int) y;
   3    #63          return (int) x;
        #64         }
 ** Inspect warning 126 ** End-of-file on: \SYS.$DATA.CSUBV.SOURCE
-PROGRAM-SOURCE SEARCH 'owner_id' 1/7 SCOPE 0
 ** Inspect warning 203 ** String not found
```

7. This example illustrates using the SOURCE SEARCH command with LOCATION as the file-locator.

```
-PROGRAM-SOURCE SEARCH 'owner_id' F/L LOCATION #main
#20             int owner_id = 0;
```

SOURCE SYSTEM

The SOURCE SYSTEM command directs Inspect to retrieve source files from a another system when the object file has moved but the source has not. SOURCE SYSTEM is a synonym for the SELECT SOURCE SYSTEM command.

```
SOURCE SYSTEM [\system ]
```

system

specifies the system from which Inspect should retrieve source files.

Default Value

If you do not specify a system name in a SOURCE SYSTEM command, Inspect uses the name of your current system.

Related Commands

- ADD SOURCE ASSIGN on page 6-14
- ENV on page 6-81
- SELECT SOURCE SYSTEM on page 6-169
- SOURCE ASSIGN on page 6-202

STEP

The STEP command resumes execution of the current program at the point where it was last suspended, and then suspends execution after the program has executed a certain number of units.

```
STEP [ step-spec ]


step-spec:  one of

   num-units [ code-unit ]
   IN [ num-units [ code-unit ] ]
   OUT [ num-calls ] [ PROC[S] | SUBPROC[S] ]


code-unit:  one of

   INSTRUCTION[S]   STATEMENT[S]   VERB[S]
```

step-spec

specifies how much code to execute before re-suspending execution.

num-units [*code-unit*]

specifies the step amount as a number of code units. When you use this form, STEP treats calls to other scope units as a single code unit.

The default for *code-unit* is STATEMENT. Code units map to slightly different entities in each source language. Refer to the source language sections for details regarding the size of a code unit.

`IN [ num-units [ code-unit ] ]`

specifies the step amount as a number of code units. The IN clause directs STEP to step into a called scope unit if a call occurs in that step range. The default for *num-units* is 1; the default for *code-unit* is STATEMENT.

The IN clause is invalid for PATHWAY requester programs.

`OUT [ num-calls ] [ PROC[S] | SUBPROC[S] ]`

specifies the step amount as a number of call exits; that is, how many scope unit calls to return from. The default for *num-calls* is 1; the maximum value allowed is 10.

The OUT clause is invalid for PATHWAY requester programs.

The PROCS clause specifies a TAL procedure and can be used in a subprocedure to step out of the containing procedure. The SUBPROCS clause specifies a TAL subprocedure and can be used to step out of a TAL subprocedure. If you specify neither PROCS nor SUBPROCS, STEP OUT defaults to SUBPROCS if the current location is within a subprocedure; otherwise, it defaults to PROCS.

Default Value

STATEMENT is the default with code-unit for high-level and INSTRUCTION is the default for code-unit in low-level. When you omit step-spec entirely, STEP uses 1 STATEMENT in high-level and 1 INSTRUCTION in low-level.

Usage Considerations

- Using STEP in Command Lists

  If STEP is in a command list, it must be the last command in the list.

- Code Units

  A code unit consists of the code from the beginning of one unit up to, but not including, the beginning of the next unit.

  Within a single scope unit, code units are counted in logical order of execution. Consequently, a branch to a label causes the branch target to be counted as the next unit. (The BSUB machine instruction is not a branch in this sense. Because it invokes a subprocedure, it is a calling instruction.)

  The capability to step by STATEMENTS or VERBS is a high-level capability. If you are in low-level mode, you can step only by INSTRUCTIONS.

- Repeating the Last STEP Command

  After you enter a STEP command, you can repeat it by pressing RETURN at the next Inspect prompt. This ability to repeat continues until you enter any other

Inspect command. When a debug event (such as a break event) interrupts the sequence, a RETURN continues to mean "redo the STEP."

- Breakpoints in Called Scope Units

  If you attempt to step over a call to a scope unit that contains a breakpoint, Inspect terminates the STEP command and stops in the called scope unit.

- Considerations when Using STEP IN
  - If you enter a STEP IN command and a procedure call is not within the range of the step, the command executes as though you had not specified the IN parameter.
  - If you enter a STEP IN command and step into the wrong procedure, enter a STEP OUT to return to the calling procedure.
  - If you enter a STEP IN command, execution stops when a new procedure is entered, regardless of the remaining range specified by the command.
- Considerations when using STEP OUT
  - The SUBPROCS clause prevents stepping out of the containing procedure if you issue a STEP OUT SUBPROC command and then repeat the command by pressing the RETURN key.
  - If you use the SUBPROCS clause when you are not in a TAL subprocedure, Inspect displays this warning message:

```
** Inspect warning 191 ** Current location is not a subproc
```

- If your program modified the S register and you issue a STEP OUT command when the current location is within a subprocedure, Inspect displays this warning message:

```
** Inspect warning 192 ** Unable to step subproc(s) due to 'S' register
modification
```

- When using STEP OUT, you cannot step out of the main scope unit (that is, the scope unit controlling the program). If you attempt to step out, Inspect displays the message:

```
** Inspect error 173 ** STEP OUT is not allowed from main scope unit
```

Usage Consideration for Accelerated Programs

The STEP command can behave differently when stepping an accelerated program on a TNS/R system because you will only be prompted at memory-exact points.

For more information, see Section 15, Using Inspect on a TNS/R System.

Limitation of the STEP Command

Inspect does not support stepping execution from the `throw` statement to the `catch` statement. When issuing the STEP command on the `throw` statement, the program execution resumes. To suspend the program execution when an exception is thrown, set a breakpoint on a statement in the appropriate catch block.

For example,

```
1.  void func (int x) {
2.    try {
3.     if (x == 0)
4.        throw "exception";
5.     cout << x;
6.     }
7.    catch(...) {
8.      cout << "In Catch block";
9.    }
10.  }
```

If you issue the STEP command on line 4 of the above program, the program execution resumes. To suspend the program execution, set a breakpoint at line 8.

Related Commands

- BREAK on page 6-19
- CLEAR on page 6-27
- RESUME on page 6-158

STOP

The STOP command stops one or more programs, removing the stopped programs from the program list.

If the program is a save file, the STOP command also closes the save file. If the program is a process or PATHWAY server, and if it is the only program being debugged, the STOP command terminates the Inspect session after stopping the program.

Note. The STOP command is invalid for PATHWAY requester programs; you must use PATHCOM to stop them.

```
STOP [ * | program ]


program:   one of

   program-number
   program-name
   cpu,pin
```

`*`

specifies all programs in the program list.

`program`

specifies a program using one of several formats. *`program`* identifies the program by its program number (as shown by the LIST PROGRAM command). *`program`* identifies the program by its name (as shown by the LIST PROGRAM command). *`program`* identifies the program by its process ID (CPU number and process number).

Default Value

If you enter the STOP command without any parameter, Inspect stops the current program.

Usage Considerations

- The STOP command can only stop programs on the program list; you cannot use it to stop any process.
- The Inspect STOP command stops only the primary process of a fault-tolerant process pair; it does not stop the backup process. If you need to stop both the primary and backup simultaneously, use the TACL command STOP.

Related Commands

- ADD PROGRAM on page 6-10
- HOLD on page 6-93
- RESUME on page 6-158
- SELECT PROGRAM on page 6-167

Example

This example illustrates the starting of a second process in debugging mode. First, the Inspect command PAUSE is used to ignore the previously existing process while issuing Inspect commands for the second process. Then the Inspect command STOP is used to terminate the second process. Inspect automatically prompts for commands for the earlier process:

```
_$UT_HIGH
-$UT-COMMENT Press BREAK key and pause to start second process
-$UT-{ BREAK key pressed }
5>RUND OBJECT /NAME $UT2/
-$UT-PAUSE
   :
   :
-$UT2-COMMENT stopping $UT2 gives prompt for remaining $UT
-$UT2-STOP
-$UT- COMMENT pause or press BREAK key to start another process
```

SYSTEM

The SYSTEM command sets the default system for expanding file names as operands to Inspect commands.

```
SYSTEM [ \system ]
```

system

is an HP NonStop system name. A system name always begins with a backslash and has one to seven additional alphanumeric characters; the first character after the backslash must be alphabetic.

Default Value

If you omit system, Inspect sets the default system to the host system of the Inspect command terminal.

Usage Consideration

Inspect does not use the default system name you set using the SYSTEM command to expand names of source files. To change the default system name for source files, use the SELECT SOURCE SYSTEM command.

Related Commands

- ENV on page 6-81
- VOLUME on page 6-223

TERM

The TERM command alters Inspect's command terminal by changing the home terminal of the Inspect process to another terminal or a process. All programs being debugged retain their original home terminal.

```
TERM { terminal | process }

terminal:

     [ \system. ] $term-name

process:

   [ \system. ] $name
```

terminal

specifies a terminal by its device name.

process

specifies a process name with an optional system name.

Usage Considerations

- You must enter the TERM command interactively from the keyboard; you cannot use it in:

  An OBEY file
  The INSPLOCL file
  The INSPCSTM file

- If the process you are debugging requires exclusive or extensive use of its home terminal, or if it owns the BREAK key, you can use the TERM command to direct Inspect to use another terminal as the Inspect command terminal.

Related Commands

- OBEY on page 6-152
- OUT on page 6-155
- SYSTEM on page 6-217

TIME

The TIME command enables you to obtain the current time within Inspect.

```
TIME [ /OUT/ <file> ]
```

file

specifies the file to which TIME will write its output.

Usage Considerations

- The TIME command is useful when performance measurements figures are needed against Inspect.
- The current time will be output in this form:

  `YYYY-MM-DDhh.mm.ss.cc`

TRACE

The TRACE command displays the call history of active scopes for the current program location. Inspect displays the call history sequentially, from the most recent to the oldest.

```
TRACE [ num-calls ] [ REGISTERS ] [ ARGUMENTS ]
```

num-calls

specifies the number of calls to list, beginning from the most recent call (that is, the call to the scope unit containing the current code location). If you do not specify *num-calls*, Inspect lists all outstanding calls.

REGISTERS

directs Inspect to interpret and display the TNS stack marker for each call listed.

The REGISTERS clause is invalid for PATHWAY requester programs and of limited usefulness in accelerated programs.

ARGUMENTS

directs Inspect to display the formal parameter names and actual parameter values for each call listed.

Usage Considerations

- When using the REGISTERS clause, Inspect displays each P-register value in normal Inspect form as a scope unit and code offset. Inspect displays the L register value in octal, the ENV register value decoded into mnemonics, and the space identifier (space ID). For the ENV register, the RP and CC values appear only for the first scope unit in the trace.

- If you specify ARGUMENTS, Inspect lists the formal parameter names and the actual parameter values for each call. If the scope unit has the VARIABLE or EXTENSIBLE attribute, only the supplied parameters are displayed.
- Numeric values (except the value of the L register) are displayed in the current output radix. Numbers in octal or hexadecimal notation have the % or %H prefix, respectively.
- The space identifier (space ID) is of the form:

  ```
  { UC | UL }.segment-num
  ```

  ;UC specifies that the code segment is within the user code space. UL specifies that the code segment is within the user library space. segment-num defines the particular code segment as an octal number. For more information about the space ID, see the *System Description Manual*.
- Trap Handler

  The operating system does not create a typical stack marker to describe the location of a trap. If a trap handler occurs anywhere in the stack, TRACE will show an incorrect trap location. For more information about trap handlers, see the ARMTRAP and SIGACTION_INIT_ routines in the *Guardian Procedure Calls Reference Manual*.
- Inspect lists the ENV register using these mnemonics:

CCE	Condition code equal
CCG	Condition code greater
CCL	Condition code less
CS	System code space bit is set
DS	System data space bit is set
K	Carry bit is set
LS	Library space bit is set
PRIV	Privileged bit is set
RP	Register stack pointer
T	Trap enable bit is set
V	Overflow bit is set

  For information about the ENV register, see the *System Description Manual*.
- Inspect displays all active TAL subprocedure calls (that is, subprocedures that have been entered but not yet exited). Here are considerations for TAL subprocedures:
  - If the num-calls parameter is specified, Inspect does not count calls to subprocedures as procedure levels.

   - If the REGISTERS clause is specified, Inspect does not display stack marker information for subprocedures.
   - If the ARGUMENTS clause is specified, Inspect does not display parameter names and values for subprocedures.
   - Inspect excludes subprocedure entries from the call history if symbol information for the containing procedure does not exist.
   - If your program modified the S register, Inspect might not be able to display all subprocedure entries in the call history. In this case, Inspect displays this warning message:

```
** Inspect warning 185 ** Unable to trace subproc(s) due to 'S' register
modification
```

- For TNS/R native processes, Inspect displays the PC register in the normal Inspect form as a scope name and offset. Inspect also displays the virtual frame pointer (VFP) for that frame.
- Inspect does not support debugging TNS/E native processes, but a TNS process can be held at a TNS/E code location due, for example, to a memory access breakpoint (MAB), a stop event, or an abend event.

  The TRACE output on TNS/E systems will not show any TNS/E native frames. If a TNS process is held at a TNS/E native address, the first line of the TRACE output shows “Unknown TNS/E Address.”

  To view both TNS and TNS/E frames, you must use Visual Inspect for TNS/E systems.

Related Commands

- LIST PROGRAM on page 6-137
- SELECT PROGRAM on page 6-167

Examples

1. When using the REGISTERS clause, the TRACE command displays the L register, ENV register, and space ID information in parentheses:

```
-TALOBJ-TRACE REGISTERS
Num Lang Location   (Registers)
  0  TAL #APPLY.#537(BNK)  (L=%21023,RP=7,CCE,K,T,UC.0)
  1  TAL #PARSER^HIGH.#240(MAIN)  (L=%20730,T,UC.0)
  2  TAL #MAIN.#168(MAIN)  (L=%20547,T,UC.0)
```

2. This example shows a TRACE command without clauses. When an entry in the call history has only the notation SYSTEM CODE or SYSTEM LIB and an octal

value, or if an entry is shown as a block name and offset with no source line location, no symbols were available in that procedure; for example:

```
-PRG-TRACE
Num Lang Location
  0  CBL #ATTEMPTDISPLAY.#1.4(QUEENSCS)
  1      #APPLY + %17
  2  CBL #QUEENSCO.#57(QUEENSCS)
```

3. A TRACE command with both REGISTERS and ARGUMENTS requested (in either order) lists the heading line first; then for each scope unit with a symbol table it lists:

   - The symbolic information about the called routine.
   - The argument values (limited to 35 characters), as if requested in a DISPLAY command.

   The next example shows a COBOL program UPSHIFT calling a COBOL program TAILOR to shift lowercase letters to uppercase. The three arguments are IN-ARRAY, MAX-I, and OUT-ARRAY. Each of the two arrays is 25 characters long.

```
-COBOBJ-TRACE REGISTERS ARGUMENTS
Num Lang Location   (Registers)  (Arguments)
  0  CBL #TAILOR.#462(UTLIB)  (L=%1014,RP=7,CCG,K,UC.0)
IN-ARRAY =
  A[1] = "delREY                   "
MAX-I = 15, OUT-ARRAY =
  B[1] = "DELREY                   "
  1   CBL  #UPSHIFT.#320(UPSHIFT)  (L=%756,RP=7,CCG,K,UC.0)
```

4. This example shows how call history entries for subprocedures differ from entries for procedures. The following TRACE command shows that the procedure MAIN called the subprocedure SPROC1, which called the subprocedure SPROC2. SPROC2 then called the procedure PROC1, which called the subprocedure SPROC3. SPROC3 is the currently executing subprocedure in PROC1:

```
-TALOBJ-TRACE
Num Lang Location
     TAL   .SPROC3:  #PROC1.#150(SOURCE)
  0  TAL  #PROC1.#200(SOURCE)
     TAL   .SPROC2:  #MAIN.#50(SOURCE)
     TAL   .SPROC1:  #MAIN.#40(SOURCE)
  1  TAL  #MAIN.#80(SOURCE)
```

   Note that the subprocedures do not have a scope number. Also, note that the subprocedure names are prefixed with a period rather than a pound sign.

5. This example shows TRACE and TRACE ARGUMENTS within a Pascal private procedure. The SOURCE command shows the procedure specification. The first

TRACE command shows the calling stack. The second TRACE command includes the ARGUMENT clause and shows the name and value of the parameters passed.

```
-P008OBJ0-SOURCE
   #64
   #65
   #66
   #67
1 *#68         procedure proc1( g1 : boolean; var g2 : boolean;
   #69                          c1 : char;    var c2 : char;
   #70                          b1 : byte;    var b2 : byte;
   #71                          i1 : integer; var i2 : integer;
   #72                          u1 : cardinal;var u2 : cardinal;
   #73                          d1 : longint; var d2 : longint;
-P008OBJ0-
   #74                          r1 : real;    var r2 : real;
   #75                          l1 : longreal;var l2 : longreal;
   #76                          s1 : colors;  var s2 : colors;
   #77                          e1 : fruit;   var e2 : fruit);
   #78
   #79         {*


   #80          *      Local variables:
   #81          *}
   #82
   #83         var
-P008OBJ0-TRACE
Num Lang Location
  0  PAS #P008SRC1.PROC1.#68(P008SRC1)
  1  PAS #P008SRC1.#251(P008SRC1)
-P008OBJ0-TRACE ARGUMENT
Num Lang Location   (Arguments)
  0  PAS #P008SRC1.PROC1.#68(P008SRC1)
G1 =  F, G2 =  F, C1 = ?0, C2 = "y", B1 = 0, B2 = 126, I1 = 650, I2 = 651,
U1 = 60000, U2 = 60001, D1 = 87542, D2 = 87543, R1 =  7.5, R2 =  7.6, L1 =  8.8,
L2 =  8.9,  = [ GREEN RED ], S2 = [ BLUE GREEN ], E1 = APPLES, E2 = BANANAS
  1  PAS #P008SRC1.#251(P008SRC1)
```

VOLUME

The VOLUME command sets the default volume and subvolume for expansion of any file names.

```
VOLUME { $volume              }
       { [ $volume. ] subvol }
```

volume

is a volume name. A volume name always begins with a dollar sign and has one to seven additional alphanumeric characters; the first character after the dollar sign must be alphabetic.

subvol

is a subvolume name. A subvolume name has one to eight alphanumeric characters; the first character must be alphabetic.

Default Value

If the volume/subvolume specification is omitted, the current Guardian subvol is changed to your initial Guardian subvolume.

Usage Considerations

- Note that the VOLUME command requires a parameter; it cannot be used without parameters to restore original defaults.
- If Inspect is started using the command interpreter command Inspect, the default volume and subvolume in Inspect will be the command interpreter defaults. In all other cases— such as RUND and Debug, the default volume and subvolume will be the logon defaults of the creator of the program being debugged.

Related Commands

- ENV on page 6-81
- SYSTEM on page 6-217

XC

The XC (“eXecute Command”) command reissues a command line from the history buffer.

```
XC | ! [ command-line-specifier ]

command-line-specifier:   one of

   pos-num
   neg-num
   search-text
   "search-text"
```

command-line-specifier

specifies which command line from the history buffer to reissue.

pos-num

is a positive integer that refers to the command-line number in the history buffer that you want to reissue.

neg-num

is a negative number that refers to a command line in the history buffer relative to the current command line.

`search-text`

is the most recent command line in the history buffer that begins with the text you specify. You need to specify only as many characters as necessary to identify the command line uniquely.

`"search-text"`

is like *`search-text`* except that the text may be anywhere in the line.

Default Value

If you do not specify a command line, Inspect reissues the last command line in the history buffer (excluding FA, FB, FC, XC, and !).

Usage Consideration

The XC command re-executes a command without storing it into the history buffer.

Related Commands

- FC on page 6-84
- HISTORY on page 6-93
- LIST HISTORY on page 6-135

7 Low-Level Inspect

Low-Level Inspect Commands

Low-level Inspect is similar to Debug, and this section discusses the major differences between the two debuggers. For more information on Debug for NonStop systems, see the *Debug Manual.* Table 7-1 shows how low-level Inspect commands correspond to Debug commands.

Table 7-1. Machine-Level Inspect Commands

Inspect Command	Debug Equivalent	Description
A	A	Displays data and registers in ASCII
B	B	Sets a code breakpoint
BM	BM	Sets a data breakpoint
C	C	Clears a code breakpoint
CM	CM	Clears a data breakpoint
D	D	Displays data and registers
F	F	Shows status of files
FN	FN	Searches memory for a number
HIGH	-	Returns to high-level Inspect
I	I	Displays data and registers in ICODE
M	M	Modifies data and registers
P	P	Pauses Inspect
R	R	Resumes program execution
S	S	Stops the current program
T	T	Displays a trace listing of stack markers
VQ	VQ	Changes extended segment
?	?	Displays segment ID
=	=	Computes and displays a value

Low-level Inspect supports all high-level commands except:

```
BREAK      CLEAR      DISPLAY      MODIFY      SCOPE
```

Syntax of Low-Level Command Elements

Low-level Inspect command syntax is based on the syntax of Debug commands. Inspect, however, allows symbolic references to code or data blocks in the break and clear commands. Inspect supports Debug syntax for referencing specific code and library segments in a multiple code segment program.

For a detailed syntax description of each of the low-level Inspect commands, use the low-level Inspect HELP command.

Symbolic References

In the B (break) and C (clear) commands, low-level Inspect allows symbolic references to a procedure name or a data block name:

```
B #block-name + nnnn
```

or

```
C #block-name + nnnn
```

where *nnnn* is the offset in words from block-name.

These symbolic references allow you to use a compilation listing without a load map to determine the program-relative address of the breakpoint.

No other symbolic references can be used in the low-level Inspect command syntax.

Multiple Code Segment Programs

For multiple code segment programs, you must specify the code segment where the address is located, in addition to the address. To specify a code segment, use the following address modes:

```
{ UC | UL } [ .segment-number, ]
```

UC indicates that the address is in the user code space. UL indicates that the address is in the user library space. Segment-number defines the particular code segment within the user space; it must be a number in the range zero through 31 decimal. If you omit the segment number, Inspect uses zero.

```
C
```

indicates an address in the current code segment (user code space or user library space).

For example, if *#block-name* is the name of a procedure in the first code segment of the user code space, the command to set a breakpoint at this location has the format:

```
B  UC.1, #block-name + nnnn
```

In a multiple code segment program, if you specify a code block name and you omit the address mode, Inspect assumes the code segment to be the same as that of the code block. If the address expression includes more than one code block reference, the first code block you specify determines the code segment.

I and S Suffixes

In low-level command syntax, the suffixes I and S always mean “indirect” and “string indirect” respectively; they do not mean “instructions” and “statements.”

Expressions in Low-Level Inspect

The syntax you use to enter an expression in low-level Inspect is based on Debug expression syntax. Here is the syntax you use to create expressions in low-level Inspect.

```
   expression:

value [ operator value ]


operator: one of

   *   /   <<   >>   +   -

value: one of

   ( expression )
   'ASCII-character ASCII-character
   #code-block
   #data-block
   number [ .number ]
   register


number is:

   [ + | - ] [ # ] integer


register: one of

   P    E    L    S
   R0   R1   R2   R3   R4   R5   R6   R7
   RA   RB   RC   RD   RE   RF   RG   RH
```

number

The options of *number* have the following meanings:

+ denotes a positive value

- denotes a negative value

# indicates a decimal, not octal, integer

If you do not use any of these options, Inspect interprets *`number`* as a positive integer in octal notation.

`number` `[ .`*`number`* `]`

specifies left and right words of a doubleword.

Using Low-Level Inspect

When using low-level Inspect, you need to know how low-level Inspect differs from high-level Inspect and how low-level Inspect differs from Debug. The following subsections discuss the differences.

Differences Between Low-Level and High-Level Inspect

The following subsections highlight the differences between low-level and high-level Inspect.

Default Radix

The high-level radix default is decimal; the low-level default is octal. Remember that these defaults differ when switching between low and high levels. You can use the SHOW RADIX command to check the current default radix.

Code Offset Units

The low-level default for code units is word instructions. The high-level default for code units varies with the particular language.

The T Command

The low-level T (Trace) command includes an indication of the code segment for each stack frame if the program has multiple code or library segments; for example:

```
021021:  150551  000202  020743     #PARSER + %12670I   UC.2
020741:  047317  000200  020730     #MA^PARSER + %166I  UC.0
020726:  047063  000200  020547     #MA^MAIN + %255I    UC.0
```

The low-level T command differs slightly from the high-level TRACE command. The high-level command includes an entry for the current scope unit; the low-level command does not.

Differences Between Low-Level Inspect and Debug

The following subsections highlight the differences between low-level Inspect and Debug.

- Low-level Inspect does not support the DEBUG ALL parameter when setting breakpoints.
- Low-level Inspect allows B (break) and C (clear) to refer to breakpoints by block name, representing the base of a program unit.
- Low-level Inspect allows you to write the output of a display to a disk file or to a nondisk file. Debug allows you to write only to a nondisk file.

Example of low-level Inspect commands:

```
D /OUT \sys.$vol.subvol.file/ 0,10     (Disk file)
D /OUT $s.#lp3/ 0,10                   (Nondisk file)
```

Example of Debug commands:

```
D0,10,$s.#lp3                          (Nondisk file)
```

- Debug allows you to modify code locations. You cannot modify code locations with either low-level or high-level Inspect.
- The low-level Inspect commands D and M display extended addresses rather than 16-bit word addresses when the memory resides in an extended data segment. For example, this D command displays memory from an extended data segment:

```
_OBJECT_D 2000000, 4
00002000000:  000000  000000  000000  000000
```

The A Command

The low-level A command without parameters provides an interpretation of the ENV register and an indication of the current code segment.

The D Command

The D command in low-level Inspect differs from the D command in Debug. Here is the syntax you use in Inspect.

```
D [ [ unit ] address [ , amount ] ] [ : base ]
  [ register                      ]

unit:  one of

  B  W  D  F
```

```
amount: one of

  num
  T width * height

base:  one of

  A  B  D  H  I  O  X  #

register:  one of

  P   E   L   S
  R0  R1  R2  R3  R4  R5  R6  R7
  RA  RB  RC  RD  RE  RF  RG  RH
```

The = Command

In Debug, the = command supports these conversion modes:

#	Decimal
A	ASCII
B	Binary
I	ICODE
E	Environmental register (flags, RP setting)

In low-level Inspect, the = command supports these additional modes:

D	Decimal
H	Hexadecimal
O	Octal
X	Hexadecimal
ENV	Same format as E (environmental register)
C	Current code segment (code block plus offset)
UC [.seg-num,]	User code segment (code block plus offset)
UL [.seg-num,]	User library segment (code block plus offset)

When you use the C, UC, or UL conversion modes, Inspect displays the value using all location formats: STATEMENTS OFFSET, LINES FILE ALL OFFSET, and INSTRUCTIONS; for example:

```
_PROG_= 500 : UC.0
     = #MAIN.6 + %11I, #MAIN.#16.41($VOL.SVOL.PRGSRC) + %11I, #MAIN + %45I
```

When you use the E or ENV conversion modes, Inspect translates and displays the value as the stack marker ENV register; for example:

```
_PROG_= 301 : env
    = (L=0,RP=1,CCG,K,T) UC.1
```

Default Volume and Subvolume

In both high and low levels, the default volume and subvolume for an Inspect process started by the debugging facility are from your logon defaults, even if you have issued a command interpreter VOLUME command to change your session defaults. If you started Inspect with a command interpreter RUN INSPECT command, Inspect uses the current session defaults.

8 Using Inspect With C

Starting to Debug a C Program

When you start a C program, the C library performs certain start-up operations before your program begins executing. To execute the start-up code and get to your program, set a breakpoint in your program then resume execution; for example:

```
-COBJ- BREAK #main
-COBJ- RESUME
```

Scope Units and Scope Paths

C has only one type of scope unit: the function. When debugging a program written in C, you must specify a function name whenever an Inspect command expects a scope unit.

Use this syntax to identify C scope paths in Inspect:

```
scope-path:
    #function
```

Code Locations

Here is the syntax you use to identify C code locations in Inspect.

```
code-location:
  { scope-path             } [FROM module ] [ offset ]...
  { [scope-path.]code-spec }

code-spec:  one of
  function
  label
  statement-number
  #line-number [ (source-file) ]

offset:
{ + | - } num [ code-unit ]

code-unit:  one of
  INSTRUCTION[S]
  STATEMENT[S]
  VERB[S]
```

scope-path

specifies the function containing the code location. When followed by a code offset, scope-path specifies the base of the function; otherwise, it specifies the primary entry point of the function.

[*scope-path*.] *code-spec*

specifies a named or numbered location in the function defined by the given scope path (or the current scope path if you omit the scope path).

function

specifies the primary entry point of the function. *function* must be the same as the function named in scope-path (or the current scope path).

label

specifies the statement following a given label in the source code.

statement-number

specifies the statement beginning at the given statement number. To see statement numbers, use the SOURCE command after you have set your location format to statements. For more information, see SET LOCATION FORMAT on page 6-175.

#*line-number* [(*source-file*)]

specifies the statement beginning at a given line number in the source file.

( ) qualifies the line number by the source file containing it. Use this option only if the source code for the given function is in more than one file.

`FROM` *`module`*

specifies the module in which the function containing the code location is defined. In HP C, the module name is the file name of the module's base source file (the file specified as input to the HP C compiler). Use the FROM clause only if you have two functions of the same name.

`offset`

specifies an offset from the code location defined by the preceding options. A positive offset denotes code following the specified code location; a negative offset denotes code preceding the specified code location. The amount to offset is specified as a given number of units. If you omit the unit specifier, Inspect selects a default unit of STATEMENT for C programs. Inspect code units correspond to C code units as follows:

INSTRUCTION	Specifies a machine-code instruction in compiled C program.
STATEMENT	Specifies a C statement.
VERB	Specifies a C statement, as does STATEMENT.

Usage Considerations

- Low-level code locations

  Low-level Inspect recognizes function names, but does not use any other symbol information. In low-level Inspect, therefore, you can only use code locations of the form:

  ```
  #scope-path [ code-offset ]
  ```

  This form represents an offset from the code base of a function. Also, the code unit of *`code-offset`* in low-level Inspect is always INSTRUCTION.

- High-level code locations

  High-level Inspect recognizes function names as does low-level Inspect, but it also uses the symbol information created when you compile a function with the SYMBOLS pragma. Therefore, code locations in high-level Inspect can include label identifiers or line numbers.

- Specifying code locations by label

  You can use a C label as the code reference in a code location. However, because Inspect also accepts scope units as code references, a conflict arises if a label's identifier is the same as the identifier for its containing function. Inspect interprets the identifier as a reference to the function, not to the label. Consequently, you must specify the code location of the label by its statement number, its line number, or its instruction offset.

- Specifying code locations by line number

  If no statement begins at the line number you specify, Inspect issues this warning:

```
** Inspect warning 117 ** A subsequent line number is assumed: line-number
```

  Inspect then uses the statement starting at the given line number. If more than one statement begins on the line you specify, Inspect uses the start of the first statement.

- The STATEMENT code unit and C statements

  Inspect recognizes these as statements:

  - Simple C statements that are not part of a compound or composite statement
  - C statements in a compound statement, delimited by braces
  - The parts of a composite C statement (*if*, *for*, and so on)

- Using the FROM Clause in a Command List

  You can use the FROM clause only once in a command list.

Examples

Given a function named `cfunct` that contains a labeled statement named `error_fix`, you can specify these code locations:

Code Location	Specifies
`#cfunct`	The primary entry point of the function `cfunct`.
`#cfunct.error_fix`	The code immediately following the label `error_fix`.

These example code locations assume the current scope path #cfunct:

Code Location	Specifies
`cfunct`	The primary entry point of the function `cfunct`.
`error_fix = 3S`	Three statements past the label `error_fix`.

Data Locations

Here is the syntax you use to identify C data locations in Inspect.

```
data-location:
   [ scope-path [ (instance) ] . ] data-reference
   [ #data-block.                ]

instance:
   [ + | - ] integer
data-reference:  one of

   identifier
   data-reference '[' subscript-range ']'
   data-reference.identifier
   data-reference->identifier
   *data-reference

subscript-range:
   expression [ :expression ]
```

scope-path [(*instance*)]

specifies the function containing the data item.

(*instance*)

identifies a specific activation of the data item's parent function. You should specify an instance only when you want to identify a local data item in a recursive function.

#data-block

specifies the global data block containing the object specified by *data-reference*. In HP C, the name of a module's global data block is a circumflex (^) followed by the file name of the module's base source file (the file specified as input to the HP C compiler). For example, the name of the global data block for the module compiled from the file MODULE1 is ^MODULE1.

You should specify a global data block only when you have two global objects of the same name in two different modules; neither module can be declared extern.

data-reference

specifies the data item using C syntax. The recursion in the definition of *data-reference* enables you to refer to complex C data structures.

identifier

specifies a simple or pointer object. When used in the DISPLAY command, *identifier* can also be the name of a structured object; *identifier* then specifies the entire object. When used in the INFO IDENTIFIER command, *identifier* can also be the name of a structured object or user-defined data type; *identifier* then specifies the entire object or the type definition.

data-reference '[' *subscript-range* ']'

specifies an array object.

subscript-range

specifies the subscript of an array element or the subscript range of a group of array elements.

data-reference.identifier

specifies a field of a structure object.

data-reference->identifier

specifies a field of a structure referenced using a structure pointer.

**data-reference*

specifies the value referenced by a pointer object.

Default Values

If you do not specify *scope-path*, Inspect uses the current scope path.

Usage Considerations

- You must compile with the SYMBOLS pragma to use *data-location*.
- A data location must specify a data object in an active scope unit.

Examples

Given a C function named `cfunct` that contains the object declaration `int bytes`, you can have these data location expressions:

Data Location	**Specifies**
`#cfunct.nbytes`	The most recent instance of `nbytes`.
`#cfunct(-1).nbytes`	The second-most recent instance of `nbytes`.
`#cfunct(1).nbytes`	The oldest (first) instance of `nbytes`.

Expressions

Here is the syntax you use to create C expressions in Inspect.

```
expression:  one of
   primary
   *expression
   &expression
   -expression
   !expression
   ~expression
   expression binary-op expression

primary:  one of
   data-location
   constant
   string
   ( expression )

binary-op:  one of
   *     /     %     +     -     >>
   <<    <     >     <=    >=    ==
   !=    &     ^     |     &&    ||
```

Usage Considerations

- Operator precedence is the same as that defined in C.
- Inspect does not support the C comma expression and operator; for example:

  `x = 4, x + 2`
- Inspect does not support the C conditional expression and operator, which requires three operands; for example:

  `a ? b : c`
- Inspect does not support the C assignment operators (=, +=, =, and so on).
- You can perform an arithmetic operation on two character constants.
- When you use a string constant in an arithmetic expression, Inspect evaluates the string constant as a character constant.
- Inspect supports type letters such as U (unsigned) and L (long).

C Data Types and Inspect

The following subsections discuss how Inspect handles and presents various C data types.

Bit Fields

Inspect can access C bit fields for display or for expression evaluation. For example, in these Inspect session the identifiers are binary objects:

```
-COBJ-DISPLAY (tstor)                  ;COMMENT Display value
 0
-COBJ-DISPLAY (tstor | tstand          ;COMMENT Bitwise OR
1
-COBJ-DISPLAY (tstand & b)             ;COMMENT Bitwise AND
1
-COBJ-DISPLAY (tstand ^ b)             ;COMMENT Bitwise XOR
0
```

Arrays

When you use an array name in an expression, Inspect interprets the array name as a pointer value. However, when you display an unsubscripted array name, Inspect displays a pointer value and the contents of the array.

For example, assume this C array declaration:

```
int z[10]
```

Inspect would process references to this array in these ways:

- If z appears in an expression, the value of z is the address of array element `z[0]`.
- If z appears as a display item, Inspect displays the address of array element `z[0]` and the contents of array z.

Structure Pointers

If you display a structure pointer but do not also select a field within the referenced structure, Inspect displays the whole structure. To select a field within the referenced structure, use the C arrow operator; for example:

```
-COBJ-DISPLAY structptr->field
```

This example shows how to display a structure and how to display the value of the pointer to the structure:

```
-COBJ-DISPLAY *structptr    ;COMMENT - displays the structure
-COBJ-DISPLAY structptr     ;COMMENT - displays the pointer value
```

Self-Referential Structures

This example displays two elements in a self-referential structure, using the C declarations:

```
struct tnode {            /* the basic node */
  char *word;             /* points to text */
  struct tnode *left;     /* left child     */
  struct tnode *right;    /* right child    */
 };

struct tnode s, *sp;
```

This example displays the structure s:

```
-COBJ-DISPLAY s
S =
  WORD = the value of word
  LEFT = pointer value
  RIGHT = pointer value
```

This example displays the data to which sp points, using tnode as a template.

```
-COBJ-DISPLAY *sp
  WORD = value of word
  LEFT = pointer value
  RIGHT = pointer value
```

Note that if you display sp instead of *sp, Inspect displays the address of sp instead of the data to which it points.

Unions

To access a particular union member, you must explicitly qualify that member. This qualification determines which field of the union Inspect accesses (which field type is actually accessed). In a display item, if you do not explicitly define the union member, Inspect displays the first member.

This examples show how to display union members, using the C declaration:

```
union u_tag {
  int ival;
  float fval;
  char *pval;
} uval;
```

In this examples, the member is explicitly defined so that the value selected will be processed:

```
-COBJ-DISPLAY uval.ival      ;COMMENT integer value displayed
UVAL.IVAL = integer value

-COBJ-DISPLAY uval.fval      ;COMMENT floating point value
UVAL.FVAL = floating point value

-COBJ-DISPLAY uval.pval      ;COMMENT pointer type
UVAL.PVAL = pointer to string
```

In this example, Inspect displays all members because no member is specified:

```
-COBJ-DISPLAY uval
UVAL =
  IVAL = integer value
  FVAL = floating point value
  PVAL = pointer to string
```

Inspect Enhancements and Restrictions for C

The following subsections discuss certain differences between programming with C and using Inspect to debug C programs.

Uppercase and Lowercase Letters

For C identifiers, Inspect distinguishes between uppercase and lowercase letters. In a multi-language environment, however, Inspect requires that you represent the data in the form that the particular language demands. For example, if C is the language of the current scope, any reference to TAL must be in uppercase. If a TAL identifier is entered in lowercase, Inspect will not upshift the identifier for resolution; it remains undefined. For example, if you are inspecting a C routine, you can display the TAL variables VAR_A and VAR^B by entering:

```
--DISPLAY VAR_A, VAR^B
```

Defining Objects in Block Structure

When you define a block of objects by putting them after the left brace that introduces a compound (block) statement, Inspect only resolves object references that are not duplicates. If there are duplicates, they are ambiguous to Inspect; therefore, Inspect displays the error message:

```
** Inspect error 98 ** Qualification required to resolve ambiguous reference:
identifier
```

Command Usage Guidelines for C Programmers

Guidelines for C programmers using Inspect are arranged alphabetically by Inspect command name. Not all commands are listed.

BREAK

- A breakpoint set at the entry point to a function will occur before any initialization. If you set a breakpoint at an entry point, you should enter a STEP 1 STATEMENT command when Inspect stops there.

DISPLAY

- The default for numeric value conversion in C, both for input and output, is DECIMAL. If you want to use DISPLAY for a quick calculation when you are in a C environment, and you want to enter octal numbers, you must either prepare for it with a SET RADIX OCTAL command or preface each octal value with a leading zero.

HELP

- You can ask for help on the definitions of Inspect command parameters. Therefore, you can find out what Inspect recognizes as a C data location, a C code location, or a C expression.

INFO IDENTIFIER

- In addition to providing attribute information for code and data locations, Inspect can provide information about macros made using `#define`. When you request the attributes of a macro, Inspect displays the replacement text associated with the macro.

INFO OPENS

- C programs begin execution with these three files already open:

`stdin`	Standard input
`stdout`	Standard output
`stderr`	Standard error

  These files are part of the run-time environment for C.

SCOPE

- If you have identifiers of the same name in different scope units, be sure that you qualify the identifiers enough for Inspect to distinguish them.

SET RADIX

- Even if you set your input radix to hexadecimal, you must still prefix a hexadecimal value with 0x or 0X if its first digit is aboveþ9; otherwise, Inspect interprets the value as an identifier.

STEP

- The STEP command defaults to STATEMENTS if no code-unit is specified. All other instances of STATEMENTS and INSTRUCTIONS in using Inspect with C (all code-location offsets as used in the BREAK command, for example) default to INSTRUCTIONS if neither is specified.
- The STEP command requires caution if `switch` statements or `for` loops are in the path. The stepping behavior of these two statements is unexpected.

  Recall that a `switch` selects one statement from a set of statements, depending on the value of a numeric expression. A STEP of one statement from the beginning of a `switch` statement takes you to the end of the entire `switch` statement. A subsequent STEP of one statement will take you to the selected `case` or `default` statement.

  If your process is at the beginning of a single-statement `for` loop (the loop body is a single statement, not a block), entering STEP 1S gets you to the beginning of the single statement, and entering STEP 2S completes execution of the loop.

9 Using Inspect With C++

Starting to Debug a C++ Program

When you start a C++ program, the C++ library performs certain start-up operations before your program begins executing. To execute the start-up code and get to your program, set a breakpoint in your program then resume execution; for example:

```
-COBJ- BREAK #main
-COBJ- RESUME
```

Scope Units and Scope Paths

C++ has three types of scope units: function, class, and object, of which, Inspect supports only functions. When debugging a program written in C++, you must specify a function name whenever an Inspect command expects a scope unit.

Here is the syntax you use to identify C++ scope paths in Inspect.

```
scope-path:

   #function
```

Note. The class name is included in the member function.

Code Locations

Here is the syntax you use to identify C++ code locations in Inspect.

```
code-location:

   { scope-path             } [ FROM module ] [ offset ]...
   { [scope-path.]code-spec }

code-spec:  one of
   function
   label
   statement-number
   #line-number [ (source-file) ]
   class::function

offset:
   { + | - } num [ code-unit ]

code-unit:  one of
   INSTRUCTION[S]
   STATEMENT[S]
   VERB[S]
```

scope-path

specifies the function containing the code location. When followed by a code offset, scope-path specifies the base of the function; otherwise, it specifies the primary entry point of the function.

[*scope-path*.] *code-spec*

specifies a named or numbered location in the function defined by the given scope path (or the current scope path if you omit the scope path).

function

specifies the primary entry point of the function. *function* must be the same as the function named in *scope-path* (or the current scope path).

label

specifies the statement following a given label in the source code.

statement-number

specifies the statement beginning at the given statement number. To see statement numbers, use the SOURCE command after you have set your location format to statements. For more information, see SET LOCATION FORMAT on page 6-175.

`#`*`line-number`* `[ (`*`source-file`*`) ]`

specifies the statement beginning at a given line number in the source file.

( ) qualifies the line number by the source file containing it. Use this option only if the source code for the given function is in more than one file.

`FROM` *`module`*

specifies the module in which the function containing the code location is defined. In HP C++, the module name is the file name of the module's base source file (the file specified as input to the HP C++ compiler). The FROM clause can be used to restrict the number of ambiguous functions. Inspect will continue to prompt you to resolve the ambiguity if there is more than one matching function. In C++, the FROM clause can still be used to specify non-static overloaded functions.

`offset`

specifies an offset from the code location defined by the preceding options. A positive offset denotes code following the specified code location; a negative offset denotes code preceding the specified code location. The amount to offset is specified as a given number of units. If you omit the unit specifier, Inspect selects a default unit of STATEMENT for C++ programs. Inspect code units correspond to C++ code units as follows:

INSTRUCTION	Specifies a machine-code instruction in compiled C++ program.
STATEMENT	Specifies a C++ statement.
VERB	Specifies a C++ statement, as does STATEMENT.

Usage Considerations

- Low-level code locations

  Low-level Inspect recognizes function names, but does not use any other symbol information. In low-level Inspect, therefore, you can only use code locations of the form:

  `#`*`scope-path`* `[` *`code-offset`* `]`

  This form represents an offset from the code base of a function. Also, the code unit of *`code-offset`* in low-level Inspect is always INSTRUCTION.

- High-level code locations

  High-level Inspect recognizes function names as does low-level Inspect, but it also uses the symbol information created when you compile a function with the SYMBOLS pragma. Therefore, code locations in high-level Inspect can include label identifiers or line numbers.

- Specifying code locations by label

  You can use a C++ label as the code reference in a code location. However, because Inspect also accepts scope units as code references, a conflict arises if a label's identifier is the same as the identifier for its containing function. Inspect interprets the identifier as a reference to the function, not to the label. Consequently, you must specify the code location of the label by its statement number, its line number, or its instruction offset.

- Specifying code locations by line number

  If no statement begins at the line number you specify, Inspect issues this warning:

```
** Inspect warning 117 ** A subsequent line number is assumed: line-number
```

  Inspect then uses the statement starting at the given line number. If more than one statement begins on the line you specify, Inspect uses the start of the first statement.

- The STATEMENT code unit and C++ statements

  Inspect recognizes these as statements:

  - Simple C++ statements that are not part of a compound or composite statement
  - C++ statements in a compound statement, delimited by braces
  - The parts of a composite C++ statement (if, for, and so on)

- Using the FROM Clause in a Command List

  You can use the FROM clause only once in a command list.

Examples

Given a function named `class::funct` that contains a labeled statement named `error_fix`, you can specify these code locations:

Code Location	**Specifies**
`#class::funct`	The primary entry point of the function `class::funct.`
`#class::funct.error_fix`	The code immediately following the label `error_fix`.

These example code locations assume the current scope path `#class::funct`:

Code Location	**Specifies**
`class::funct`	The primary entry point of the function `class::funct`.
`error_fix + 3S`	Three statements past the label `error_fix`.

Data Locations

Here is the syntax you use to identify C++ data locations in Inspect.

```
data-location:
   [ scope-path [ (instance) ] . ] data-reference
   [ #data-block.                ]

instance:
   [ + | - ] integer
data-reference:  one of

   identifier
   data-reference '[' subscript-range ']'
   data-reference.identifier
   data-reference->identifier
   *data-reference

subscript-range:
   expression [ :expression ]
```

scope-path [(*instance*)]

specifies the function containing the data item.

(*instance*)

identifies a specific activation of the data item’s parent function. You should specify an instance only when you want to identify a local data item in a recursive function.

#data-block

specifies the global data block containing the object specified by data-reference. In HP C++, the name of a module’s global data block is a circumflex (^) followed by the file name of the module’s base source file (the file specified as input to Cfront). For example, the name of the global data block for the module compiled from the file MODULE1 is ^MODULE1.

You should specify a global data block only when you have two global objects of the same name in two different modules; neither data block can be declared `extern`.

data-reference

specifies the data item using C++ syntax. The recursion in the definition of *data-reference* enables you to refer to complex C++ data structures.

identifier

specifies a simple or pointer object. When used in the DISPLAY command, *identifier* can also be the name of a structured object; *identifier* then specifies the entire object. When used in the INFO IDENTIFIER command, `identifier` can also be the name of a structured object or user-defined data type; `identifier` then specifies the entire object or the type definition.

`data-reference '[' subscript-range ']'`

specifies an array object.

`subscript-range`

specifies the subscript of an array element or the subscript range of a group of array elements.

`data-reference.identifier`

specifies a field of a structure object.

`data-reference->identifier`

specifies a field of a structure referenced using a structure pointer.

`*data-reference`

specifies the value referenced by a pointer object.

Default Values

If you do not specify scope-path, Inspect uses the current scope path.

Usage Considerations

- You must compile with the SYMBOLS pragma to use `data-location`.
- A data location must specify a data object in an active scope unit.

Examples

Given a C++ function named `class::funct` that contains the object declaration `int nbytes`, you can have these data location expressions:

Data Location	**Specifies**
`#class::funct.nbytes`	The most recent instance of `nbytes`.
`#class::funct(-1).nbytes`	The second-most recent instance of `nbytes`.
`#class::funct(1).nbytes`	The oldest (first) instance of `nbytes`.

Expressions

Here is the syntax you use to create C++ expressions in Inspect.

```
expression:  one of
   primary
   *expression
   &expression
   -expression
   !expression
   ~expression
   expression binary-op expression

primary:  one of
   data-location
   constant
   string
   ( expression )

binary-op:  one of
   *     /     %     +     -     >>
   <<    <     >     <=    >=    ==
   !=    &     ^     |     &&    ||
```

Usage Considerations

- Expression handling within Inspect does not support overloaded or redefined operators.
- Operator precedence for C++ is the same as that defined in C.
- Inspect does not support the C++ comma expression and operator; for example:

  ```
  x = 4, x + 2
  ```

- Inspect does not support the C++ conditional expression and operator, which requires three operands; for example:

  ```
  a ? b : c
  ```

- Inspect does not support the C++ assignment operators (=, +=, =, and so on).
- You can perform an arithmetic operation on two character constants.
- When you use a string constant in an arithmetic expression, Inspect evaluates the string constant as a character constant.
- Inspect supports type letters such as U (unsigned) and L (long).

C++ Data Types and Inspect

The following subsections discuss how Inspect handles and presents various C++ data types.

Bit Fields

Inspect can access C bit fields for display or for expression evaluation. For example, in these Inspect session the identifiers are binary objects:

```
-COBJ-DISPLAY (tstor)                ;COMMENT Display value
 0
-COBJ-DISPLAY (tstor | tstand        ;COMMENT Bitwise OR
1
-COBJ-DISPLAY (tstand & b)           ;COMMENT Bitwise AND
1
-COBJ-DISPLAY (tstand ^ b)           ;COMMENT Bitwise XOR
0
```

Arrays

When you use an array name in an expression, Inspect interprets the array name as a pointer value. However, when you display an unsubscripted array name, Inspect displays a pointer value and the contents of the array. For example, assume this C array declaration:

```
int z[10]
```

Inspect would process references to this array in these ways:

- If z appears in an expression, the value of z is the address of array element `z[0]`.
- If z appears as a display item, Inspect displays the address of array element `z[0]` and the contents of array z.

Structure Pointers

If you display a structure pointer but do not also select a field within the referenced structure, Inspect displays the whole structure. To select a field within the referenced structure, use the C++ arrow operator; for example:

```
-COBJ-DISPLAY this->class::field
```

This example shows how to display a structure and how to display the value of the pointer to the structure:

```
-COBJ-DISPLAY *this   COMMENT - displays the object
-COBJ-DISPLAY this    COMMENT - displays the pointer value
```

Unions

To access a particular union member, you must explicitly qualify that member. This qualification determines which field of the union Inspect accesses (which field type is actually accessed). In a display item, if you do not explicitly define the union member, Inspect displays the first member.

Inspect Enhancements and Restrictions for C++

The following subsections discuss certain differences between programming with C++ and using Inspect to debug C++ programs.

Uppercase and Lowercase Letters

For C++ identifiers, Inspect distinguishes between uppercase and lowercase letters. In a multi-language environment, however, Inspect requires that you represent the data in the form that the particular language demands. For example, if C++ is the language of the current scope, any reference to TAL must be in uppercase.

If a TAL identifier is entered in lowercase, Inspect will not upshift the identifier for resolution; it remains undefined. For example, if you are inspecting a C++ function, you can display the TAL variables VAR_A and VAR^B by entering:

```
--DISPLAY VAR_A, VAR^B
```

Defining Objects in Block Structure

When you define a block of objects by putting them after the left brace that introduces a compound (block) statement, Inspect only resolves object references that are not duplicates. If there are duplicates, they are ambiguous to Inspect; therefore, Inspect displays the error message:

```
** Inspect error 98 ** Qualification required to resolve ambiguous reference:
identifier
```

Overloaded Functions

Inspect will detect when an ordinary function or member function is overloaded, and prompt you to resolve the ambiguity. For example:

```
-EDEMO-MATCH SCOPE funct
Program Procedures:
funct(void)
funct(int)
funct(int,int)
-EDEMO-BREAK #funct
Specified scope is ambiguous
  [1] funct(void)
  [2] funct(int)
  [3] funct(int,int)
Which scope do you mean ([1], 2, ...)? 2
Num Type Subtype Location
  1 Code         #funct(int).#11
-EDEMO-LIST BREAK
Num Type Subtype Location
  1 Code         #funct(int).#11
-EDEMO-
```

The FROM clause can be used to specify which functions you want. In this example, `ctest1` is the source file for `funct(void)` and `ctest2` is the source file for `funct (int)` and `funct (int,int)`.

```
-EDEMO-BREAK #funct FROM ctest1
 Num Type Subtype Location
   1 Code         #funct(void).#4
```

Note that because there is only one function named `#funct` in source file `ctest1`, Inspect can uniquely identify the function.

Overloaded Operators

Expression handling within Inspect does not support overloaded operators. Inspect does allow you to set breakpoints in the overloaded operator. For example:

```
-ETST1-BREAK #Cat::operator=
Num Type Subtype Location
  1 Code         #Cat::operator=(Cat&).#26
-ETST1-LIST BREAK
Num Type Subtype Location
  1 Code         #Cat::operator=(Cat&).#26
-ETST1-
```

Static Data

Static data items have global scope in C++, which means that they are not allocated within each object. Inspect will only display the data members that are local to that object. This is source code followed by an example of displaying an object with static data.

```
class Example_Class {
  public:
    int  local_var1;
    int  local_var2;
    static int global_var;
     void member_func(void);
};
int Example_Class::global_var;
```

```
-PROGRAM-DISPLAY Example_Object
Example_Object =
  Example_Class::local_var1 = 99
  Example_Class::local_var2 = 45
```

The MATCH SCOPE command may be used to find the names of static data members. For example:

```
-PROGRAM-MATCH SCOPE Example_Class::*
Program Procedures:Example_Class::member_func(void)

Program Data:
Example_Class::global_var
```

The `this` Pointer

When the current scope is a nonstatic member function, local member data items must be qualified with the “`this`” pointer. For example:

```
-PROGRAM-DISPLAY this->Example_Class::local_var_2
Example_Class.Example_Class::local_var_2 = 45
```

To display all the variables local to the object that invoked the member function, use the command:

```
-PROGRAM-DISPLAY *this
Example_Class =
   Example_Class::local_var_1 = 99
  Example_Class::local_var_2 = 45
```

Usage Considerations

The HP C++ translator, Cfront, translates a C++ program into an equivalent C program. Translation results in some restrictions with debugging C++ programs.

These restrictions are:

- “References” are not supported. Cfront implements references as pointers. To display a reference to an object, use the “*” operator to dereference it. In this example, “i” is an integer and “r” a reference to “i”:

```
-PROGRAM-DISPLAY r
r = 52
-PROGRAM-DISPLAY i
i = -10
-PROGRAM-DISPLAY *r
short = -10
```

- Inspect does not support pointers to data members.
- Breakpoints are not supported for:
    - Inlined functions
    - Member functions of locally defined classes
- Cfront treats structures like classes. Inspect requires class-like syntax to access fields of structures. The source below was used to generate this example.

  ```
  struct Example  {
    int i;
    int j;
  };
  struct Example var;
  ```

```
-PROGRAM-DISPLAY var
var =
   Example::i = 10
  Example::j = 52
```

- Inspect allows you to shorten data member names for C++ objects. Using the previous example:

```
-PROGRAM-DISPLAY var.i
var.i = 10
-PROGRAM-DISPLAY var.Example::i
var.Example::i = 10
```

- Inspect adds an “_” character to conversion operator functions and the new and delete operators. For example, to set a breakpoint at the new operator type:

  ```
  -PROGRAM-BREAK #operator_new
  ```

- Note that Cfront may add intermediate fields in classes, thereby making it difficult to display them.

Command Usage Guidelines for C++ Programmers

The following guidelines for C++ programmers using Inspect are arranged alphabetically by Inspect command name. Not all commands are listed.

BREAK

- A breakpoint set at the entry point to a function will occur before any initialization. If you set a breakpoint at an entry point, you should enter a step over initialization before accessing any data. Alternatively, a break set at the first line of a function will have the desired effect.

DISPLAY

- The default for numeric value conversion in C++, both for input and output, is DECIMAL. If you want to use DISPLAY for a quick calculation when you are in a C++ environment, and you want to enter octal numbers, you must either prepare for it with a SET RADIX OCTAL command or preface each octal value with a leading zero.

HELP

- You can ask for help on the definitions of Inspect command parameters. Therefore, you can find out what Inspect recognizes as a C++ data location, a C++ code location, or a C++ expression.

INFO IDENTIFIER

- In addition to providing attribute information for code and data locations, Inspect can provide information about macros made using `#define`. When you request the attributes of a macro, Inspect displays the replacement text associated with the macro.

INFO OPENS

- C++ programs begin execution with the following three files already open:

`stdin`	Standard input
`stdout`	Standard output
`stderr`	Standard error

These files are part of the run-time environment for C++.

MATCH

- Use the SCOPE clause of the MATCH command to find all the member functions of a given class. For example:

  ```
  -PROGRAM-MATCH SCOPE Class::*
  ```

  It can also be used to find all classes that implement a given function. For example:

  ```
  -PROGRAM-MATCH SCOPE *error
  ```

SCOPE

- If you have identifiers of the same name in different scope units, be sure that you qualify the identifiers enough for Inspect to distinguish them. If you have a function named the same, Inspect will prompt you to resolve the ambiguity.
- Since scope units may be overloaded in C++, Inspect may not be able to determine the specified scope. In the case of ambiguity, you will be prompted for clarification.

SET RADIX

- Even if you set your input radix to hexadecimal, you must still prefix a hexadecimal value with 0x or 0X if its first digit is aboveþ9; otherwise, Inspect interprets the value as an identifier.

STEP

The STEP command requires caution if `switch` statements or `for` loops are in the path. The stepping behavior of these two statements is unexpected.

Recall that a `switch` selects one statement from a set of statements, depending on the value of a numeric expression. A STEP of one statement from the beginning of a `switch` statement takes you to the end of the entire `switch` statement. A subsequent STEP of one statement will take you to the selected case or `default` statement.

If your process is at the beginning of a single-statement `for` loop (the loop body is a single statement, not a block), entering STEP 1S gets you to the beginning of the single statement, and entering STEP 2S completes execution of the loop.

10 Using Inspect With COBOL and SCREEN COBOL

Note. In this section, the term COBOL refers to COBOL 74, COBOL85, and SCREEN COBOL.

Scope Units and Scope Paths

COBOL has only one type of scope unit: the program unit. When debugging a program written in COBOL, you must specify a program unit name whenever an Inspect command expects a scope unit.

Here is the syntax you use to identify COBOL scope paths in Inspect.

COBOL 74 and SCREEN COBOL

```
scope-path:
  #program-unit
```

COBOL85

```
scope-path:
   #program-unit [ .program-unit ]...
```

```
program-unit
```

is the name of a program unit.

Usage Considerations

- COMMON program units

  Inspect does not support the special scoping rules that COBOL85 provides for COMMON program units. In COBOL85, such program units are globally visible, regardless of where they are declared. In Inspect, however, you must provide the scope path to a COMMON program unit.

Code Locations

Here is the syntax you use to identify COBOL code locations in Inspect.

```
code-location:
   { scope-path              } [ offset ]...
   { [scope-path.]code-spec }

code-spec:  one of
   program-unit
   section
   paragraph [ OF section ]
   statement-number
   #line-number [ (source-file) ]

offset:
   { + | - } num [ code-unit ]

code-unit:  one of
   INSTRUCTION[S]
   STATEMENT[S]
   VERB[S]
```

scope-path

specifies the scope path to the program unit containing the code location. When used alone, *scope-path* specifies the primary entry point of the last program unit named in the scope path. When followed by a code offset, *scope-path* specifies the base of the program unit.

[*scope-path*.] *code-spec*

specifies a named or numbered location in the program unit defined by the given scope path (or the current scope path if you omit the scope path).

program-unit

specifies the primary entry point of the program unit. program-unit must be the same as the last program unit named in scope-path (or the current scope path).

section

specifies a section within the program unit.

paragraph [OF *section*]

specifies a paragraph within the program unit. If a paragraph named *paragraph* exists in two different sections of the program unit, you must use the OF clause to qualify the paragraph name.

statement-number

specifies the statement beginning at the given statement number. To see statement numbers, use the SOURCE command after you have set your location format to statements. For more information, see SET LOCATION FORMAT on page 6-175.

#*line-number* [(*source-file*)]

specifies the COBOL statement beginning at a given line number in the source file.

(*source-file*) qualifies the line number by the source file containing it. You need to use this option only if the source code for the given program unit is in more than one file.

offset

specifies an offset from the code location defined by the preceding options. A positive offset (+) denotes code following the specified code location; a negative offset (-) denotes code preceding the specified code location. The amount to offset is specified as a given number of units. If you omit the unit specifier, Inspect selects STATEMENT as the code unit for COBOL. Inspect code units correspond to COBOL code units as follows:

`INSTRUCTION`	Specifies a machine-code instruction in COBOLþ74 and COBOL85, or a pseudocode instruction in SCREEN COBOL.
`STATEMENT`	Specifies a COBOL sentence.
`VERB`	Specifies a COBOL statement.

Usage Considerations

- Low-level code locations

  Low-level Inspect recognizes program unit names but does not use any other symbol information. In low-level Inspect, therefore, you can only use code locations of the form:

  #*scope-path* [*code-offset*]

  This form represents an offset from the code base of a scope unit. Also, the code unit of *code-offset* in low-level Inspect is always INSTRUCTION.

- High-level code locations

  High-level Inspect recognizes program unit names as does low-level Inspect, but it also uses the symbol information created when you compile a program unit with the SYMBOLS directive. Therefore, code locations in high-level Inspect can include section names, paragraph names, or line numbers.

- Specifying code locations by line number

  If no COBOL statement begins at the line number you specify, Inspect issues this warning:

```
** Inspect warning 117 ** A subsequent line number is assumed: line-number
```

  Inspect then uses the statement starting at the given line number. If more than one COBOL statement begins on the line you specify, Inspect uses the start of the first statement.

COBOL 74 and SCREEN COBOL Examples

This example assume the COBOL program fragment:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. prg.
  ...
PROCEDURE DIVISION.
  SECTION. s1.
    PARAGRAPH. p1.
      ...
    PARAGRAPH. p2.
      ...
  SECTION. s2.
    PARAGRAPH. p1.
      ...END PROGRAM prg.
```

Here are some code locations:

Code Location	**Specifies**
PRG.PRG	The primary entry point of program unit PRG.
#PRG.P2	Paragraph P2 of section S1 of program unit PRG.
#PRG.P1 OF S2	Paragraph P1 of section S2 of program unit PRG. The clause OF S2 is required to distinguish between the paragraph P1 in section S1 and the one in section S2.

These code locations assume a current scope path of #PRG:

Code Location	**Specifies**
S2	Section S2 of program unit PRG.
S1 + 3S	Three COBOL sentences past the start of section S1 of program unit PRG (the fourth sentence of S1).
S1 + 3S + 2V	Two COBOL statements past three COBOL sentences past the start of section S1 of program unit PRG (the third statement of the fourth sentence of S1).

COBOL85 Examples

This example assume the COBOL85 program fragment:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. a.
    ...
    IDENTIFICATION DIVISION.
    PROGRAM-ID. b.
        ...
        IDENTIFICATION DIVISION.
        PROGRAM-ID. c.
          ...
        END PROGRAM c.
    END PROGRAM b.
END PROGRAM a.
```

Here are some code locations:

Code Location	Specifies
#A	The primary entry point of program unit A.
#A.B	The primary entry point of program unit B.
#A.A +1V	One COBOL statement past the primary entry point of program unit A. If any of the data declarations in A include a VALUE clause, the code offset +1V will perform them. Consequently, this code location specifies the first user-written sentence in the program unit.

Data Locations

Here is the syntax you use to identify COBOL data locations in Inspect.

```
data-location:
   [ scope-path. ] data-reference

data-reference:
  identifier [ OF identifier ]... [ (index [ ,index ]...) ]
index:
   expression [ :expression ]
```

scope-path

specifies the scope path to the program unit containing the data item.

data-reference

specifies a data item in the program unit defined by the given scope path (or the current scope path if none is given).

`identifier`

specifies a data item in the program unit.

`OF identifier`

specifies the group data-item in which the given data item is declared. Like COBOL, Inspect requires only enough qualification to identify the data item uniquely.

`index`

specifies the subscript of an array element or the subscript range of a group of array elements.

Usage Considerations

- Any COBOL data location to which you refer must be a data item defined in a program unit that was compiled with the SYMBOLS directive. The location can be anywhere in the DATA DIVISION of an active scope unit.
- You must qualify a data reference enough to identify a data item uniquely. If a data reference identifies more than one data item, Inspect issues an error message stating that the reference is ambiguous (except when you are using the INFO IDENTIFIER command; in this case, Inspect reports all occurrences of the identifier in the scope unit).
- Inspect does not support the examination or modification of items in the SCREEN SECTION of a SCREEN COBOL program.
- You can use the regular COBOL subscripts to refer to a specific item in a table. Inspect also allows you to specify a range of subscripts for DISPLAY or MODIFY commands. For example:

  ```
  -COBOBJ-DISPLAY payscale-table(1:3, 1:4)
  ```
- Inspect requires that subscripts be separated by commas; in COBOL, commas are optional.
- A data location cannot include mnemonic names.
- Inspect does not support the examination or modification of COBOL items whose usage is INDEX. For example, if the table X-TAB is indexed by X, Inspect will reject commands such as:

  ```
  -COBOBJ-DISPLAY x-tab(x)
  -COBOBJ-MODIFY x
  -COBOBJ-MODIFY zz = z + x-tab (x)
  ```

Inspect, however, allows you to use subscripts instead of indices, so it will accept commands such as:

```
-COBOBJ-DISPLAY x-tab(1), x-tab(5:9)
-COBOBJ-MODIFY x-tab(r:r+9)  ;COMMENT R is not USAGE INDEX
-COBOBJ-BREAK x-handling+3S IF x-tab(1) > 2145
```

Examples

This example assume these COBOL declarations:

```
01 master-file-rec.
   03 mfr-employee-name.
      05 lastname          PICTURE X(20).
      05 firstname         PICTURE X(20).
   03 mfr-employee-number  PICTURE 99999.
      88 hired-before-merger VALUE IS 1 THROUGH 35929.
      88 hired-after-merger  VALUE IS 35930 THROUGH 99999.
   ...

01 paytable.
   03 payscale             OCCURS 20 TIMES
                           PICTURE 999V99
                           USAGE IS COMPUTATIONAL.
   ...77 week-of-year            PICTURE 99.
```

Here are some data locations:

Data Location	**Specifies**
MFR-EMPLOYEE-NAME	MFR-EMPLOYEE-NAME OF MASTER-FILE-REC
FIRSTNAME	FIRSTNAME OF MFREMPLOYEENAME OF MASTERFILEREC
WEEK-OF-YEAR	WEEK-OF-YEAR
PAYSCALE (3)	PAYSCALE OF PAYTABLE (3)

Special Registers

If you compile a COBOL 74 or COBOL85 program with or without the WITH DEBUGGING MODE clause in the SOURCE COMPUTER paragraph, these data items are available:

PROGRAM-STATUS	Indicates the outcome of the STARTBACKUP, OPEN, CLOSE, and CHECKPOINT NonStop Facility operations. For more information, see the discussion of fault-tolerant processes in the *COBOL85 for NonStop Systems Manual*.
GUARDIAN-ERR	Indicates the latest file system error number. For more information about run-time diagnostic messages, see the *COBOL85 for NonStop Systems Manual*.

However, if you compile your program using the WITH DEBUGGING MODE clause in the SOURCE COMPUTER paragraph, these data item is available:

DEBUG-ITEM	Is a part of the ANSI Standard debugging mechanism. For more information, see the description of the USE statement in the *COBOL85 for NonStop Systems Manual*.

DEBUG-ITEM is mainly used for non-interactive debugging, but the PROGRAM-STATUS and GUARDIAN-ERR registers can be helpful in debugging with Inspect.

Expressions

Here is the syntax you use to create COBOL expressions in Inspect.

```
expression:
   condition [ { AND | OR } condition ]...

condition:
   [ NOT ] { simple-exp [ rel-op simple-exp ]... }
           { level-88-condition                   }

rel-op:
   [ NOT ] {  >  =  <  GREATER  EQUAL  LESS  }

simple-exp:
   [ + | - ] term [ { + | - } term ]...

term:
   factor [ { * | / } factor ]...

factor:
   primary [ **primary ]

primary:  one of
   data-location
   "@data-location"
   number
   ( expression )
```

Usage Considerations

- Operator precedence is the same as the precedence defined in COBOL.
- When you use the minus sign (-) as an operator, you must delimit it with spaces.
- Inspect does not support these in expressions:

  Class conditions, such as ALPHABETIC and NUMERIC
  Sign conditions, such as NEGATIVE and POSITIVE
  Figurative constants, such as HIGH-VALUES, SPACES, and so on
  Abbreviated compound conditions, such as (A > B OR C)

COBOL Data Types and Inspect

The following subsections discuss how Inspect handles and presents various COBOL data types.

Record Types

This example assume these COBOL record declarations:

```
01 office-record.
    05 room-number     PIC X9(4)  VALUE "A4294".
    05 address         PIC X(35)  VALUE "555 12th Street".
    05 square-footage  PIC 999V99 VALUE 113.40
                       USAGE IS COMPUTATIONAL.
    05 phone-number    PIC 9(10)  VALUE 9113732411.01
 lumber-table.
    03 thickness OCCURS 2 TIMES.
        05 width OCCURS 6 TIMES.
            07 price PICTURE 99V99.
```

This example displays the entire record, including the names of the record fields:

```
-COBOLOBJ-DISPLAY office-record
OFFICE-RECORD =
  ROOM-NUMBER = "A4294"
  ADDRESS = "555 12th Street
  SQUARE-FOOTAGE =  113.40
  PHONE-NUMBER =  9113732411.
```

This example displays the record as a contiguous block of characters. The ?0 ?0 represent octal values that cannot be displayed as characters:

```
-COBOLOBJ-DISPLAY office-record WHOLE
OFFICE-RECORD = "A4294555 12th Street                    " ?0 ?0 ",L9113732411"
```

This example displays the record without field names:

```
-COBOLOBJ-DISPLAY office-record PLAIN
A4294
555 12th Street
113.4
9113732411.
```

This example displays the record as a contiguous block of characters, suppressing the record name:

```
-COBOLOBJ-DISPLAY office-record WHOLE PLAIN
A4294555 12th Street                     ?0 ?0,L9113732411
```

This example shows the internal attributes of some record fields:

```
-COBOBJ-INFO IDENTIFIER price
PRICE: VARIABLE
storage^info:
TYPE=NUM UNSIGN, ELEMENT LEN=8 BITS, UNIT SIZE=4 ELEMENTS,
SCALE=2
```

```
access^info:
'L'+%22S WORDS
dimension^info:
[1:2,1:6]
structure^info:
PARENT=WIDTH

-COBOBJ-INFO IDENTIFIER width
WIDTH: VARIABLE
storage^info:
TYPE=CHAR, ELEMENT LEN=8 BITS, UNIT SIZE=4 ELEMENTS
access^info:
'L'+%22S WORDS
dimension^info:
[1:2,1:6]
structure^info:
PARENT=THICKNESS,CHILD=PRICE

-COBOBJ-INFO IDENTIFIER thickness
THICKNESS: VARIABLE
storage^info:
TYPE=CHAR, ELEMENT LEN=8 BITS, UNIT SIZE=24 ELEMENTS
access^info:
'L'+%22S WORDS
dimension^info:
[1:2]
structure^info:
PARENT=LUMBER-TABLE,CHILD=WIDTH
```

Inspect Enhancements and Restrictions for SCREEN COBOL

Because the TCP, not DMON, provides the execution control services needed to debug SCREEN COBOL programs, certain Inspect commands and options are not applicable to SCREEN COBOL. They are:

- Data breakpoints
- The BREAK command's ABEND, BACKUP, READ, and STOP clauses
- Code locations in the DISPLAY command
- The DISPLAY command's AS, FOR, and WHOLE clauses
- The INFO OBJECT, INFO OPENS, INFO SCOPE, and INFO SEGMENTS commands
- The LOW command
- The MATCH SCOPE command
- The MODIFY command's WHOLE clause
- The SAVE command

- The STEP IN and STEP OUT commands
- The STOP command
- The TRACE command's REGISTERS clause

Command Usage Guidelines for COBOL Programmers

The following guidelines for COBOL programmers using Inspect are arranged alphabetically by Inspect command name. Not all commands are listed.

BREAK

- You can set a breakpoint at a line number, a paragraph, a paragraph of a section, a section, or the program entry point. You can qualify the following names with a scope name.

  You can also set a breakpoint at an offset from a line number, a paragraph, a paragraph of a section, a section, or the program entry point. Offsets are given as a number of sentences (Inspect calls them STATEMENTS), statements (Inspect calls them VERBS), or machine instructions or pseudo instructions (Inspect calls them INSTRUCTIONS).
- For SCREEN COBOL, you can set a breakpoint only at the start of an inactive program unit. Also, when you set a breakpoint at the entry of a program unit, Inspect does not check your SCREEN COBOL library to ensure that the given program unit exists. Consequently, if you set a breakpoint at a nonexistent program unit, Inspect does not display a warning message.
- When you run COBOL-compiled processes, you can also set and clear a single breakpoint in the data area. Because a data breakpoint monitors only one word in the data area, any operation that changes the contents of that word (alone, or as part of a group item) will trigger the Inspect break event.
- A break set at the entry point to a program unit will occur before any initialization (VALUE clause actions). If you set a breakpoint at a program unit entry point, you should enter a STEP 1 VERB command when Inspect stops there. Alternatively, a break set at the first paragraph of a program unit will have the desired effect.
- If you set a breakpoint at ABEND and your COBOL program terminates abnormally, you will find that your program is executing system code instead of user code. To access code or data in your program, you must qualify code and data references to include a scope path. Alternatively, you can change the current scope path with the SCOPE command.
- If you have a paragraph name in your program called ABEND, entering BREAK ABEND causes Inspect to set a breakpoint at the paragraph, not at abnormal program end.

- If you set a data breakpoint within a group item, and your program moves a value to the group, the break event occurs before the move operation is complete because COBOL always uses byte moves for group items.

DISPLAY

- The DISPLAY command does not display FILLER items in records unless you use the WHOLE clause.
- There are significant differences between the PIC clause of the DISPLAY command and the COBOL PICTURE clause. The PICTURE clause is part of a data item's definition, while the PIC clause is simply a template for formatting a data item. Also, the PIC clause is case-sensitive, excludes constructs such as parenthesized repetition counts, and does not perform floating replacement. In fact, the only characters that have special meaning in the mask string you provide in DISPLAY's PIC clause are:

  V (uppercase V only
  Z (uppercase Z only)
  9

  The formatter displays all other characters (including “z” and “v”) exactly as they appear in the mask string.

  In addition, the mask string must be enclosed in either quotes (") or apostrophes ('). To include a quote in a quote-delimited mask string, use a pair of quotes. To include an apostrophe in an apostrophe-delimited mask string, use a pair of apostrophes.
- The DISPLAY command can display COBOL items that have been declared in the Extended-Storage Section. To display these items, use the standard DISPLAY command syntax described in Section 6, High-Level Inspect Commands.

HELP

You can ask for help on the definitions of Inspect command parameters. Therefore, you can find out what Inspect recognizes as a COBOL data location, a COBOL code location, or a COBOL expression.

INFO IDENTIFIER

- If you request a listing of the attribute of an unqualified data name, Inspect will list all possible instances of that data name in the current scope.
- The data types reported by Inspect are not designated in terms of any one language. Therefore, COBOL computational items of up to four digits are marked BIN SIGN, while COBOL alphanumeric items are marked CHAR. A number of data types in Inspect are directly COBOL oriented, such as NUM LD EM (numeric, sign leading, embedded) and NUM TR SP (numeric, sign trailing, separate).

- When you request information about all identifiers in a program unit, Inspect lists the identifiers in the reverse order of declaration for COBOL 74 and COBOL85; Inspect lists the identifiers in alphabetical order for SCREENþCOBOL.

MODIFY

- A MODIFY command operating on an edited field does not edit. Inspect functions just as the VALUE clause does, inserting a character value into the data item unchanged.
- When you use a MODIFY command to assign a value to a group item, you must use the MODIFY WHOLE form. The value you provide in the command must be a quoted string value.
- When you specify the name of a group item in a MODIFY WHOLE command, Inspect requires the new value to be in the command list. Inspect will not prompt for a new value.
- When an alphanumeric data item in a COBOL program is being modified, the & operator can be used to concatenate bytes with a string. The values can be in binary, octal, or hex format.
- The MODIFY command does not restrict COMPUTATIONAL items to 18-digit values; it will allow you to store a 19-digit value. However, COBOL cannot manipulate COMPUTATIONAL values in excess of 18 digits.
- The MODIFY command can modify COBOL items that have been declared in the Extended-Storage Section. To modify these items, use the standard MODIFY command syntax described in Section 6, High-Level Inspect Commands.

SCOPE

- For COBOL, the scope of an identifier is always the program-ID of the program unit in which the identifier is declared.
- If you have identifiers of the same name in different program units, be sure that you qualify the identifiers enough for Inspect to distinguish them.

SET RADIX

- Even if you set your input radix to hexadecimal, you must still prefix a hexadecimal value with zero or “%H” if its first digit is above nine; otherwise, Inspect interprets the value as an identifier.

STEP

- The STEP command considers COBOL CALL, ENTER, and PERFORM statements as one verb. If you step through a portion of the program and come to a PERFORM statement, a subsequent STEP 1 VERB command executes the entire PERFORM scope (including any TIMES or UNTIL constraints) before you are prompted again.

If you want to step through the code within the PERFORM scope, you must set a breakpoint at the label that marks the beginning of the PERFORM scope. You can then enter a STEP command when Inspect reports the break event.

If you are stepping within a performed or called scope, and control returns to the statement following the PERFORM, CALL, or ENTER, stepping does not cease. In other words, you can step out of a PERFORM, CALL, or ENTER statement, but you cannot step into one.

11 Using Inspect With FORTRAN

Scope Units and Scope Paths

FORTRAN has three types of scope units: program, subroutine, and function. When debugging a program written in FORTRAN, you must specify one of these types whenever an Inspect command expects a scope unit.

Here is the syntax you use to identify FORTRAN scope paths in Inspect.

```
scope-path:
   #scope-unit

scope-unit:   one of
   program
   subroutine
   function
```

Usage Consideration

- BLOCK DATA

  Note that BLOCK DATA subprograms are not listed as a type of FORTRAN scope unit. You cannot directly access the data initialization in a BLOCK DATA subprogram; however, you can access the initialized common data through the variables you associate with common data in a COMMON statement.

Code Locations

.Here is the syntax you use to identify FORTRAN code locations in Inspect.

```
code-location:
   { scope-path             } [ offset ]...
   { [scope-path.]code-spec }

code-spec:  one of
   scope-unit
   statement-function
   statement label
   entry point
   statement-number
   #line-number [ (source-file) ]

offset:
   { + | - } num [ code-unit ]

code-unit:  one of
   INSTRUCTION[S]
   STATEMENT[S]
   VERB[S]
```

scope-path

specifies the program unit containing the code location. When used alone, *scope-path* specifies the primary entry point of the last program unit. When followed by a code offset, *scope-path* specifies the base of the program unit.

[*scope-path*.] *code-spec*

specifies a named or numbered location in the program unit defined by the given scope path (or the current scope path if you omit the scope path).

scope-unit

specifies the primary entry point of the program unit. *scope-unit* must be the same as the program unit named in *scope-path* (or the current scope path).

statement-function

specifies a statement function.

statement-label

specifies the statement following a statement label. The SET LOCATION FORMAT command is used to differentiate between statement-label and statement-number. See Usage Considerations on page 11-3.

`entry-point`

specifies an entry point.

`statement-number`

specifies the statement beginning at the given statement number. The SET LOCATION FORMAT command is used to differentiate between statement-label and statement-number. For more information, see Usage Considerations.

To see statement numbers, use the SOURCE command after you have set your location format to statements. For more information, see SET LOCATION FORMAT on page 6-175.

`#`*`line-number`*` [ (`*`source-file`*`) ]`

specifies the statement beginning at a given line number in the source file.

(*`source-file`*) qualifies the line number by the source file containing it. You need to use this option only if the source code for the given program unit is in more than one file.

`offset`

specifies an offset from the code location defined by the preceding options. A positive offset (+) denotes code following the specified code location; a negative offset (-) denotes code preceding the specified code location. The amount to offset is specified by a given number of units. If you omit the unit specifier, Inspect selects STATEMENT as the code unit for FORTRAN. Inspect code units correspond to FORTRAN code units as follows:

INSTRUCTION	Specifies a machine-code instruction.
STATEMENT	Specifies a FORTRAN statement.
VERB	Specifies a FORTRAN statement, as does STATEMENT

Usage Considerations

- Inspect does not recognize variables used as labels (by the ASSIGN statement) as code locations.
- Certain labeled or unlabeled FORTRAN statements might not generate any code; they mark locations in the program. Among these statements are CONTINUE, END, and ENDIF. Because such statements do not generate any code, Inspect does not recognize them as statements.

  A CONTINUE statement that ends a DO loop does generate code because the code for the DO loop control is located there. (For more information about the STEP command, see Command Usage Guidelines for FORTRAN Programmers on page 11-11.)

- Specifying code locations by line number

  If no statement begins at the line number you specify, Inspect issues this warning:

```
** Inspect error 117 ** A subsequent line number is assumed: line-number
```

  Inspect then uses the statement starting at the given line number. If more than one statement begins on the line you specify, Inspect uses the start of the first statement.

- Specifying statement-number or statement-label.

  In order for Inspect to determine if a number given for a code location is a statement-number or statement-label, the setting of LOCATION FORMAT is used.

  The current setting can be checked by using the SHOW LOCATION FORMAT command. If the LOCATION FORMAT includes STATEMENTS, the number is interpreted to mean statement-number. If STATEMENTS does not appear in the LOCATION FORMAT, then the number is interpreted to mean statement-label.

Examples

- This example shows the effect of using number with combinations of LOCATION FORMAT with and without STATEMENT.

  When the LOCATION FORMAT includes STATEMENT, the number in the code location is interpreted to mean statement 20.

```
-FOROBJ-SET LOCATION FORMAT LINE, STATEMENT
-FOROBJ-SOURCE AT 20
   16   #42                  ll
 = .not. ll   17   #43                  goto 25
   18   #44                else
   19   #45                  assign 30 to next
   20   #46                  goto next
        #47                endif
   21   #48                  goto bad
        #49
   22   #50
         30      assign 50 to next
   23   #51                if ( ir .eq. 0 ) then
```

  When the LOCATION FORMAT does not include STATEMENT, the number in the code location is interpreted to mean label 20.

```
-FOROBJ-SET LOCATION FORMAT LINE
-FOROBJ-SOURCE AT 20
   #36                  goto bad
   #37                endif
   #38                  goto bad
   #39
   #40         20      ll = .false.
   #41         25      if ( .not. ll) then
   #42                   ll = .not. ll
```

```
     #43                       goto 25
     #44                     else
     #45                       assign 30 to next
```

- This example assume this FORTRAN program fragment in a subroutine named LOSSES:

  Here are some code locations:

Code Location	**Specifies**
#LOSSES	The primary entry point of subroutine LOSSES.
#LOSSES.100	The statement at label 100.
#LOSSES.100 -1S	The statement before label 100.
#LOSSES.THRESHOLD	The statement function THRESHOLD.

```
    SUBROUTINE LOSSES
    COMMON /TALLIES/ COLLAPSE, OSCILLATE
    ...
    THRESHOLD(I) = MIN (I*1.E-2, X(I,1))
    ...
    DO 120 I = 1, 26
       DO 110 J = 1, 26
100       MARGIN (I) = MARGIN (I) - X(I,J)
          IF ( MARGIN (I) .LT. THRESHOLD (Z) ) THEN
             MARGIN (I) = 0
             COLLAPSE = COLLAPSE + 1
          END IF
110       CONTINUE
120    CONTINUE
```

Data Locations

Here is the syntax you use to identify FORTRAN data locations in Inspect.

```
data-location:
   [ scope-path [ (instance) ] . ] data-reference

instance:
   [ + | - ] num

data-reference:  one of

   identifier
   data-reference ( index [ ,index ]...)
   data-reference^identifier

subscript-range:
   expression [ :expression ]
```

scope-path [(*instance*)]

specifies the program unit containing the data item.

(*instance*) identifies a specific activation of the data item's parent program unit. You need to specify an instance only when you want to identify a local data item in a recursive program unit.

data-reference

specifies the data item using FORTRAN syntax. The recursion in the definition of data-reference enables you to refer to complex FORTRAN data structures.

identifier

specifies a simple variable. When used in the INFO IDENTIFIER or DISPLAY command, identifier can also be the name of a structured variable; identifier then specifies the entire variable.

data-reference (*index* [,*index*]...)

specifies an array variable.

index

specifies the subscript of an array element or the subscript range of a group of array elements.

data-reference^identifier

specifies a field of a record variable.

Default Values

If you do not specify a scope path, Inspect uses the current scope path.

When you refer to an element of a FORTRAN record, you must qualify the element name completely.

Usage Considerations

- Any FORTRAN data location to which you refer must be a data item defined in a program unit that was compiled with the SYMBOLS directive. The location can be anywhere in an active scope unit.
- If you declare a variable but never refer to it, FORTRAN does not include it in the data area. Consequently, Inspect has no information about the variable and will issue a message.
- You cannot set a data breakpoint to detect the use of a FORTRAN FORMAT statement by a READ or WRITE statement.

- Subscripting data locations in Inspect is the same as in FORTRAN. For example, this DDL structure requires three subscripts when you refer to the data item SUBFIELD:

  ```
  RECORD REC.
    03 SUBREC OCCURS ...
       05 FIELD OCCURS ...
          07 SUBFIELD OCCURS ...
  ```

  A data reference to SUBFIELD must include subscripts for all of its parent data items:

  ```
  REC^SUBREC(X)^FIELD(Y)^SUBFIELD(Z)
  ```

- Statement functions are treated as subprocedures. Suppose, for example, there is an identifier POMME local to a statement function, and the program unit that contains the statement function also contains an identifier named POMME. Inspect provides no way for the user to qualify a mention of POMME (such as for DISPLAY or for setting a data breakpoint).

Examples

This example assume the FORTRAN program fragment in a subroutine named LOSSES:

```
    COMMON /TALLIES/ COLLAPSE, OSCILLATE
    ...
    THRESHOLD(I) = MIN (I*1.E-2, X(I,1))
    ...
    DO 120 I = 1, 26
       DO 110 J = 1, 26
100       MARGIN (I) = MARGIN (I) - X(I,J)
          IF ( MARGIN (I) .LT. THRESHOLD (Z) ) THEN
             MARGIN (I) = 0
             COLLAPSE = COLLAPSE + 1
          END IF
110       CONTINUE
120    CONTINUE
```

Here are some data locations:

Data Location	**Specifies**
#LOSSES.J	The loop variable J.
#LOSSES.I	The loop variable I, not the variable I in the statement function THRESHOLD. To refer to a variable in a statement function, you must find its address with the INFO IDENTIFIER command and use low-level mode to display or modify it.

These data locations assume a current scope path of #LOSSES:

Data Location	Specifies
X(5+I,4)	An element of the array X.
COLLAPSE	The variable COLLAPSE in the COMMON block TALLIES. Note that the elements of a COMMON block have a subprogram as their scope unit, not the COMMON block.

Expressions

Here is the syntax you use to create FORTRAN expressions in Inspect.

```
expression:
   condition [ bool-op condition ]...

bool-op:  one of
   .AND.  .OR.  .EQV.  .NEQV.

condition:
   [ .NOT. ] simple-exp [ rel-op simple-exp ]

rel-op:  one of
   .LT.  .LE.  .GT.  .GE.  .EQ.  .NE.

simple-exp:
   [ + | - ] term [ { + | - } term ]...

term:
   factor [ { * | / } factor ]...

factor:
   primary [ **primary ]

primary:  one of
   data-location
   constant
   ( expression )
```

Usage Considerations

- Operator precedence is the same as the precedence defined for FORTRAN.
- ASSIGN statements and the integer variables to which label values are assigned are ignored by Inspect.

FORTRAN Data Types and Inspect

The following subsections discuss how Inspect handles and presents various FORTRAN data types.

Arrays

Inspect supports all FORTRAN array types, including multidimensional arrays and arrays of records. You can use a single element, a group of elements, or an entire array as the *data-reference* part of a data location:

Data Reference	**Example**	**Where Valid**
Single element	MYARRAY(5)	In any Inspect command
Range of elements	MYARRAY(1:7)	In any Inspect command except BREAK
Entire array	MYARRAY	In the INFO IDENTIFIER and DISPLAY commands only

If you use the array name only in any command other than INFO IDENTIFIER or DISPLAY, Inspect displays the error:

```
** Inspect error 80 ** Required subscript missing: identifier
```

Examples

This example assume the FORTRAN declaration:

```
INTEGER A(10,20)
```

This example shows the internal attributes of the array:

```
-FORTOBJ-INFO IDENTIFIER a
A: VARIABLE
storage^info:
TYPE=BIN SIGN, ELEMENT LEN=16 BITS, UNIT SIZE=1 ELEMENTS
access^info:
'L'+2I+%13 WORDS
dimension^info:
[1:10,1:20]
```

This example displays a single element of the array:

```
-FORTOBJ-DISPLAY a(2,2)
A[2,2] = 202
```

This example displays a range of elements:

```
-FORTOBJ-DISPLAY a(2:6,2)
A[2,2] = 202 302 402 502 602

-FORTOBJ-DISPLAY a(2,2:6)
A[2,2] = 202
A[2,3] = 203
A[2,4] = 204
A[2,5] = 205
A[2,6] = 206
```

This example displays the entire array:

```
-FORTOBJ-DISPLAY a
A[1,1] = 101 201 301 401 501 601 701 801 901 1001
A[1,2] = 102 202 302 402 502 602 702 802 902 1002
A[1,3] = 103 203 303 403 503 603 703 803 903 1003
A[1,4] = 104 204 304 404 504 604 704 804 904 1004
A[1,5] = 105 205 305 405 505 605 705 805 905 1005
A[1,6] = 106 206 306 406 506 606 706 806 906 1006
A[1,7] = 107 207 307 407 507 607 707 807 907 1007
A[1,8] = 108 208 308 408 508 608 708 808 908 1008
A[1,9] = 109 209 309 409 509 609 709 809 909 1009
A[1,10] = 110 210 310 410 510 610 710 810 910 1010
A[1,11] = 111 211 311 411 511 611 711 811 911 1011
A[1,12] = 112 212 312 412 512 612 712 812 912 1012
A[1,13] = 113 213 313 413 513 613 713 813 913 1013
A[1,14] = 114 214 314 414 514 614 714 814 914 1014
A[1,15] = 115 215 315 415 515 615 715 815 915 1015
A[1,16] = 116 216 316 416 516 616 716 816 916 1016
A[1,17] = 117 217 317 417 517 617 717 817 917 1017
A[1,18] = 118 218 318 418 518 618 718 818 918 1018
A[1,19] = 119 219 319 419 519 619 719 819 919 1019
A[1,20] = 120 220 320 420 520 620 720 820 920 1020
```

Records

Inspect supports FORTRAN records, as shown in this example, which assume this FORTRAN declaration:

```
RECORD REC
  CHARACTER*6 DATE
  FILLER*4
  CHARACTER*6 AMOUNT
END RECORD
```

This example shows the internal attributes of the record:

```
-FORTOBJ-INFO IDENTIFIER rec
REC: VARIABLE
storage^info:
TYPE=BYTE STRUCT, ELEMENT LEN=8 BITS, UNIT SIZE=16 ELEMENTS
access^info:
'L'+%11S WORDS
structure^info:
CHILD=DATE
```

This example shows the internal attributes of one of the fields:

```
-FORTOBJ-INFO IDENTIFIER rec^date
DATE: VARIABLE
storage^info:
TYPE=CHAR, ELEMENT LEN=8 BITS, UNIT SIZE=6 ELEMENTS
access^info:
(parent)+0  WORDS
structure^info:
PARENT=REC,SIBLING=AMOUNT
```

This example displays the record using Inspect's default formatting:

```
-FORTOBJ-DISPLAY rec
REC =
  DATE = "082282"
  AMOUNT = "000405"
```

This example displays a single field of the record:

```
-FORTOBJ-DISPLAY rec^amount
REC^AMOUNT = "000405"
```

This example displays the record, suppressing the identifiers and quotes:

```
-FORTOBJ-DISPLAY rec PLAIN
082282
000405
```

This example displays the whole record, including the FILLERs:

```
-FORTOBJ-DISPLAY rec WHOLE
REC = "082282    000405"
```

This example displays the whole record, suppressing the identifiers and quotes:

```
-FORTOBJ-DISPLAY rec WHOLE PLAIN
082282
000405
```

Inspect Enhancements and Restrictions for FORTRAN

The following subsection discusses the difference between programming with FORTRAN and using Inspect to debug FORTRAN programs.

Spaces in Identifiers

Unlike FORTRAN, Inspect does not allow embedded spaces in identifiers. If your FORTRAN program includes identifiers that contain embedded spaces, you must remember to enter them without spaces in Inspect.

Command Usage Guidelines for FORTRAN Programmers

The following guidelines for FORTRAN programmers using Inspect are arranged alphabetically by Inspect command name. Not all commands are listed.

BREAK

- A break set at the entry point to a scope unit occurs before any initialization. If you set a breakpoint at a scope unit's entry point, you should enter a STEP 1S command after the breakpoint is triggered.
- Because a data breakpoint is associated with a single word, an identifier designating multiple words (such as COMPLEX, DOUBLE PRECISION, or RECORD) has only its first word marked as the breakpoint. If an EQUIVALENCE declaration allows you to read or modify a subsequent word of such a variable without affecting the primary word, the debugging facility will not signal a debug event.
- If you set a data breakpoint within a record, and your program moves a value into the entire record, the break event occurs before the move operation is complete.
- BREAK at a code location that contains a number, for example BREAK 10 or BREAK #CONTROL.10, will cause the breakpoint to be placed on the statement number, if the LOCATION FORMAT includes STATEMENT, or on the label if LOCATION FORMAT does not include statement. Use the SHOW LOCATION FORMAT to check the current setting. Use the SET LOCATION FORMAT to change the setting.

DISPLAY

- Inspect does not allow commands after a FORMAT specification in the DISPLAY command. If a FORMAT clause is present, it must be the last item in a command list.
- Inspect encloses complex values in parentheses when it displays them.

HELP

You can ask for help on the definitions of Inspect command parameters. Therefore, you can find out what Inspect recognizes as a FORTRAN data location, a FORTRAN code location, a FORTRAN expression, or a FORTRAN conditional expression.

INFO IDENTIFIER

- Inspect requires complete qualification of names of record elements. If you request a listing of the attributes of an unqualified data name, and the only instance of that name is in a record, Inspect will give you an error message. If the data name occurs in the scope twice—once as a simple variable and once as part of a record—requesting a listing of the attributes of the unqualified data name will produce an attribute listing for only the simple variable. You must qualify the name completely when you request a listing of the attributes for a record element.
- FORTRAN statement functions are treated as equivalent to TAL subprocedures. Inspect tags statement functions as being subprocedures of the procedures (program units) in which they are declared. Inspect requires full qualification of

names, so you must qualify a statement function name with its scope name if you want to request information on the attributes of a statement function.

- The data types reported by Inspect are not designated in terms of any one language. Therefore, FORTRAN integers are marked as BIN SIGN, while FORTRAN real, complex, logical, and character items are marked as REAL, COMPLEX, LOGICAL, and CHAR, respectively.
- If you request a listing of the attributes of an entire program unit, they are listed in an order determined by the FORTRAN compiler:
  - All identifiers that are not components of records are listed in alphabetical order. When a record identifier is listed, its components and their attributes are immediately listed in record-layout order.
  - All statement labels are listed in numeric order.

INFO OPENS

The suffix “F” indicates that numeric values are FORTRAN logical unit numbers. If you do not include this suffix, Inspect assumes file system numbers.

MODIFY

- Just as in FORTRAN itself, you can MODIFY a variable that is named in an EQUIVALENCE statement.
- A complex constant is represented as a pair of numeric values separated by a comma and delimited by parentheses, just as the constant would appear in a FORTRAN program.
- Record elements named FILLER cannot be explicitly modified, and an interactive MODIFY command will not prompt you for them. A MODIFY WHOLE will assign characters to all positions in a record, including those designated FILLER.

SCOPE

- In FORTRAN, the scope of an identifier is always the name of the scope unit in which the identifier is declared. COMMON blocks and statement functions are not really scope units; they refer to identifiers declared in other scope units.
- If you have identifiers of the same name in different scope units, be sure that you qualify the identifiers enough for Inspect to distinguish them.

SET RADIX

Even if you set your input radix to hexadecimal, you must still prefix a hexadecimal value with zero or “%H” if its first digit is above nine; otherwise, Inspect interprets the value as an identifier.

STEP

FORTRAN tests for continuation of a DO loop at the end of the loop. Consequently, a STEP 1S command at the beginning of a DO loop takes you to the terminal statement of the loop. A subsequent STEP 1S command takes you to the statement following the DO.

TRACE

The TRACE ARGUMENTS command shows the arguments for the primary entry point of a scope unit. If a secondary entry point has a different set of arguments, TRACE ARGUMENTS does not display them.

12 Using Inspect With Pascal

Scope Paths and Scope Units

Pascal has two types of scope units: procedure and function. When debugging a program written in Pascal, you must specify one of these types whenever an Inspect command requires or expects a scope unit.

Here is the syntax you use to identify Pascal scope paths in Inspect.

```
scope-path:

   #scope-unit [ .scope-unit ]...

scope-unit:  one of
   function
   procedure
```

Usage Guidelines

Exported procedures: If an exported procedure (a procedure declared in another compilation unit) has the same name as a module, Inspect cannot access its symbol information.

Code Locations

Here is the syntax you use to identify Pascal code locations in Inspect.

```
code-location:
  { scope-path              } [FROM module ] [ offset ]...
  { [scope-path.]code-spec }

code-spec:  one of
  scope-unit
  label
  statement-number
  #line-number [ (source-file) ]

offset:
{ + | - } num [ code-unit ]

code-unit:  one of
  INSTRUCTION[S]
  STATEMENT[S]
  VERB[S]
```

scope-path

specifies the function containing the code location. When followed by a code offset, *scope-path* specifies the base of the function; otherwise, it specifies the primary entry point of the function.

[*scope-path*.] *code-spec*

specifies a named or numbered location in the function defined by the given scope path (or the current scope path if you omit the scope path).

scope-unit

specifies the primary entry point of the scope unit. *scope-unit* must be the same as the last scope unit named in *scope-path* (or the current scope path).

label

specifies the statement following a given label in the source code.

statement-number

specifies the statement beginning at the given statement number. To see statement numbers, use the SOURCE command after you have set your location format to statements. For more information, see SET LOCATION FORMAT on page 6-175.

`#`*`line-number`* `[ (`*`source-file`*`) ]`

specifies the statement beginning at a given line number in the source file.

`( source-file`) qualifies the line number by the source file containing it. Use this option only if the source code for the given function is in more than one file.

`FROM` *`module`*

specifies the module in which the function containing the code location is defined. In Tandem Pascal, the module name is the file name of the module's base source file (the file specified as input to the Tandem Pascal compiler). Use the FROM clause only if you have two functions of the same name.

`offset`

specifies an offset from the code location defined by the preceding options. A positive offset denotes code following the specified code location; a negative offset denotes code preceding the specified code location. The amount to offset is specified as a given number of units. If you omit the unit specifier, Inspect selects a default unit of STATEMENT for Pascal programs. Inspect code units correspond to Pascal code units as follows:

INSTRUCTION	Specifies a machine-code instruction in compiled Pascal program.
STATEMENT	Specifies a Pascal statement.
VERB	Specifies a Pascal statement, as does STATEMENT.

Usage Considerations

- Low-level code locations

  Low-level Inspect recognizes function names, but does not use any other symbol information. In low-level Inspect, therefore, you can only use code locations of the form:

  `#`*`scope-path`* `[` *`code-offset`* `]`

  This form represents an offset from the code base of a function. Also, the code unit of *`code-offset`* in low-level Inspect is always INSTRUCTION.

- High-level code locations

  High-level Inspect recognizes function names as does low-level Inspect, but it also uses the symbol information created when you compile a function with the SYMBOLS pragma. Therefore, code locations in high-level Inspect can include label identifiers or line numbers.

- Specifying code locations by line number

  If no statement begins at the line number you specify, Inspect issues this warning:

```
** Inspect warning 117 ** A subsequent line number is assumed: line-number
```

Inspect then uses the statement starting at the given line number. If more than one statement begins on the line you specify, Inspect uses the start of the first statement.

- The STATEMENT code unit and Pascal statements

  Inspect recognizes these as statements:

  - Simple Pascal statements that are not part of a conditional or repetitive structured statement
  - Pascal statements in a compound statement (BEGIN...END)
  - The parts of a conditional or repetitive statement

- Using the FROM Clause in a Command List

  You can use the FROM clause only once in a command list.

Examples

This example assume this Pascal code fragment:

```
PROCEDURE a;
    ...
    PROCEDURE b;
        ...
        FUNCTION c : integer;
            ...
        END;
        ...
    END;
    ...
END;
```

Here are some code locations:

Code Location	Specifies
#A	The primary entry point of procedure A.
#A.B	The primary entry point of procedure B.
#A.B.C	The primary entry point of function C.

Data Locations

Here is the syntax you use to identify Pascal data locations in Inspect.

```
data-location:
   [ scope-path [ (instance) ] . ] data-reference
   [ #data-block.               ]

instance:
   [ + | - ] integer
data-reference:  one of

   identifier
   data-reference '[' index [ ,index]... ']'
   data-reference.identifier
   data-reference^

index:
   expression [ :expression ]
```

scope-path [(*instance*)]

specifies the scope path to the scope unit containing the data item.

(*instance*) identifies a specific activation of the data item's parent scope unit. You need to specify an instance only when you want to identify a local data item in a recursive scope unit.

#data-block

specifies the global data block containing a nonexported global variable. data-block is the name of the module containing the global variable specified by *data-reference*. The name of the data block for the main program is _MAIN_MODULE, not the program name.

You need to specify a data block only when you have two nonexported global variables of the same name in two different modules.

data-reference

specifies the data item using Pascal syntax. The recursion in the definition of *data-reference* enables you to refer to complex Pascal data structures.

identifier

specifies a simple or pointer variable. When used in the DISPLAY command, *identifier* can also be the name of a structured variable; *identifier* then specifies the entire variable. When used in the INFO IDENTIFIER command, *identifier* can also be the name of a structured variable or user-defined data type; *identifier* then specifies the entire variable or the type definition.

data-reference '[' *index* [,*index*]... ']'

specifies an array variable.

index

specifies the subscript of an array element or the subscript range of a group of array elements.

data-reference.identifier

specifies a field of a record variable.

*data-reference^*

specifies the value referenced by a pointer or buffer (file) variable.

Default Values

If you do not specify scope-path, Inspect uses the current scope path.

Usage Considerations

- Any Pascal data location to which you refer must be declared in a scope unit that was compiled with the SYMBOLS directive. The location can be anywhere in an active scope unit.

Examples

This example assume the Pascal program fragment:

```
PROGRAM nested;
    VAR y,z : integer;
    ...
    PROCEDURE outer;
        VAR a,b : integer;
        ...
        PROCEDURE inner;
            VAR c,d : integer;
            ...
        END;
    END;
END;
```

Here are some data locations:

Data Location	**Specifies**
#OUTER.INNER.C	The most recent instance of C.
#OUTER.INNER(-1).C	The second-most recent instance of C.
#OUTER.A	The most recent instance of A.
#OUTER.INNER(1).D	The oldest (first) instance of D.

Expressions

Here is the syntax you use to create Pascal expressions in Inspect.

```
expression:
   simple-exp [ rel-op simple-exp ]...

rel-op:  one of
   =  <>  <  >  <=  >=

simple-exp:   [ + | - ] term [ add-op term ]...

add-op:  one of
   +  -  OR

term:
   factor [ mult-op factor ]...

mult-op:  one of
   *  /  DIV  MOD  AND  <<  >>

factor:  one of
   data-location
   unsigned-constant
   NOT factor
   (expression)
```

Usage Considerations

- The precedence of operators is the same as the precedence defined in Pascal.
- Inspect does not support function calls in expressions.
- Inspect does not support set expressions.

Pascal Data Types and Inspect

The following subsections discuss how Inspect handles and presents various Pascal data types.

Array Types

Inspect supports all Pascal array types, including multidimensional arrays and arrays of records. You can use a single element, a group of elements, an entire array, or an array type itself as the data-reference part of a data location:

Data Reference	Example	Where Valid
Single element	MYARRAY[5]	In any Inspect command
Range of elements	MYARRAY[1:7]	In any Inspect command except BREAK
Entire array	MYARRAY	In the INFO IDENTIFIER and DISPLAY commands only
Array type	INTARRAY	In the INFO IDENTIFIER command only

If you use the array name only in any command other than INFO IDENTIFIER or DISPLAY, Inspect displays the error:

```
** Inspect error 80 ** Required subscript missing: identifier
```

Inspect performs bounds checking on Pascal arrays. If you attempt to access an element outside of the array bounds, Inspect displays the error:

```
** Inspect error 79 ** Subscript value outside of declared bounds: identifier
```

Examples

This example assume these Pascal declarations:

```
CONST  subscript = 100;

TYPE   intarray = ARRAY [1..subscript] OF INTEGER;
       chararray = PACKED ARRAY [1..subscript] OF CHAR;

VAR    a1 : intarray;
       a2 : chararray;
```

This example displays an array element and then displays both arrays:

```
-PASOBJ-DISPLAY a1[5]
A1[5] = value of fifth element

-PASOBJ-DISPLAY a1
A1 = values of all elements of a1

-PASOBJ-DISPLAY a2
A2 = values of all elements of a2 as a string of CHARs
```

Enumerated Types

Inspect supports variables of enumerated types.

Examples

This example assume these Pascal declarations:

```
TYPE
  fruit_type = (apples, oranges, bananas)

VAR  fruit : fruit_type
```

When you use an identifier of enumerated type in a DISPLAY command, Inspect displays the enumeration value as declared in the TYPE declaration. If you want Inspect to display the ordinal representation, specify a numeric format along with the identifier:

```
-PASOBJ-DISPLAY fruit
FRUIT = ORANGES

-PASOBJ-DISPLAY fruit IN DECIMAL
FRUIT = 1
```

You can modify a variable of an enumerated type using an enumeration value or an ordinal value:

```
-PASOBJ-MODIFY fruit = apples
-PASOBJ-MODIFY fruit = 0
```

The INFO IDENTIFIER command gives this information:

```
-PASOBJ-INFO IDENTIFIER fruit
FRUIT: VARIABLE
storage^info:
TYPE=DEFINED TYPE, ELEMENT LEN=8 BITS, UNIT SIZE=1 ELEMENTS
access^info:
#GLOBAL+1  WORDS
structure^info:
CHILD= FRUIT_TYPE-PASOBJ-INFO IDENTIFIER fruit_type
FRUIT_TYPE: DEFINED TYPE
access^info:
TYPE= ENUMERATION
-PASOBJ-INFO IDENTIFIER oranges
ORANGES: NAMED CONST
storage^info:
TYPE=BIN SIGN, ELEMENT LEN=16 BITS, UNIT SIZE=1 ELEMENTS
access^info:VALUE=  1
```

File Types

Inspect can access file variables; however, Inspect cannot recognize that the variable actually pertains to a file. Inspect represents a file variable as a 16bit unsigned integer.

Pointer Types

Inspect supports variables of pointer types. Inspect also supports dereferencing of pointer variables using the circumflex (^) operator.

Examples

In the following type and variable declarations, `link` is defined as a pointer type pointing to a record of type `object`. The variables base and p are declared as pointers of type link:

```
TYPE
  link = ^object;
  object = RECORD
      next : link;
      data : char
    END;
VAR
  base, p : link;
```

This example displays the pointer variable p, the record to which it points, a field of that record, and then a field in the record to which the record p^.next points:

```
-PASOBJ-DISPLAY p
P= value of the pointer

-PASOBJ-DISPLAY p^
NEXT= value of field in record to which p points
DATA= value of field in record to which p points

-PASOBJ-DISPLAY p^.next
P.NEXT= value of field in record to which p points

-PASOBJ-DISPLAY p^.next^.data
P.NEXT.DATA= value of field in next record in chain
```

Record Types

Inspect supports Pascal standard records, variant records, and free type-unions.

When you refer to a variable of a variant record type, Inspect accesses only the active fields and displays the proper values. Inspect does not display irrelevant fields.

When you refer to a field of a free type-union variable, the data type of the field determines how Inspect displays the values. If you refer to a free type-union variable without specifying a field, Inspect displays each of the possible field variants.

Examples Using a Standard Record

To access a single element of a structure, Inspect requires full name qualification. This code fragment declares a standard record:

```
TYPE
  tmprec =
     RECORD
     var1 : integer;
     var2 : integer
     END;
```

```
VAR
  record_structure : tmprec;
```

Inspect accesses records of this type as follows:

```
-PASOBJ-DISPLAY record_structure
RECORD_STRUCTURE =
  VAR1 = value of var1
  VAR2 = value of var2

-PASOBJ-DISPLAY record_structure.var1
RECORD_STRUCTURE.VAR1 = value of var1
```

Examples Using a Variant Record

This example shows implicit and explicit qualification of a variant record, assuming this Pascal program fragment:

```
TYPE
  sidetype = (offense,defense);
  positiontype = (line,back);
  team = RECORD
    CASE side : sidetype OF
      offense:
        (CASE o_position : positiontype OF
          line: (o_lineman : (TE,SE,Guard,Tackle,Center));
          back:(o_back : (Quarterback,Halfback,Fullback))
      );
      defense:
        (CASE d_position : positiontype OF
          line : (d_lineman : (End,Tackle,Nosetackle) );
          back : (d_back : (Linebacker,Safety,Cornerback))
        )
  END;

VAR  football : team;

BEGIN
  football.side := offense;
  football.o_position := back;
  football.o_back := Quarterback;
  ...
```

This example shows implicit qualification. Inspect determines which path to take by evaluating variant indicator fields as it processes the variant record:

```
-PASOBJ-DISPLAY football
FOOTBALL =
  SIDE = OFFENSE
  O_POSITION = BACK
  O_BACK = QUARTERBACK
```

This example shows explicit name qualification. Inspect follows the path given in the data location. The element can then be processed and displayed:

```
      -PASOBJ-DISPLAY football.d_backFOOTBALL.D_BACK =
      LINEBACKER
```

Examples Using a Free TypeUnion

This example shows implicit and explicit qualification of a free type-union, assuming this Pascal program fragment:

```
TYPE
  thingtype = (int,re,bool);
  thing = RECORD
    CASE thingtype OF
      int : (intval : integer);
      re  : (reval  : real);
      bool: (boolval: boolean)
  END;

VAR
  something : thing;
```

This example displays the entire record. Inspect cannot know what path to take, so it displays all paths:

```
-PASOBJ-DISPLAY something
SOMETHING.INTVAL = value displayed as an integer
SOMETHING.REVAL = value displayed as a real
SOMETHING.BOOLVAL = value displayed as a boolean
```

This example specifies an explicit path. Inspect takes this particular path to the variable:

```
-PASOBJ-DISPLAY something.boolval
SOMETHING.BOOLVAL = value displayed as a boolean
```

Set Types

Inspect allows you to use the INFO IDENTIFIER, BREAK, and DISPLAY commands with set variables.

Examples

This example assume these Pascal declarations:

```
TYPE
  colorset = (red,white,blue,yellow);
  colors = SET OF colorset;

VAR
  flag : colors;

BEGIN
```

```
  flag := (red,white,blue);
  ...
```

This example displays the set variable flag:

```
-PASOBJ-DISPLAY flag
FLAG = (RED, WHITE, BLUE)
```

This example shows the internal attributes of the objects from the previous declarations:

```
-PASOBJ-INFO IDENTIFIER flag
FLAG: VARIABLE
storage^info:
TYPE=DEFINED TYPE, ELEMENT LEN=256 BITS, UNIT SIZE=1 ELEMENTS
access^info:
#GLOBAL+1  WORDS
structure^info:
CHILD= COLORS

-PASOBJ-INFO IDENTIFIER colors
COLORS: DEFINED TYPE
access^info:
TYPE= SET, COMPONENT= COLORSET

-PASOBJ-INFO IDENTIFIER colorset
COLORSET: DEFINED TYPE
access^info:
TYPE= ENUMERATION

-PASOBJ-INFO IDENTIFIER blue
BLUE: NAMED CONST
storage^info:
TYPE=BIN SIGN, ELEMENT LEN=16 BITS, UNIT SIZE=1 ELEMENTS
access^info:
VALUE=  2
```

Subrange Types

Inspect allows a variable of subrange type as a data location parameter in any Inspect command. Inspect allows integer, boolean, character, and enumeration (scalar) subrange types.

Examples

This example assume these Pascal declarations:

```
TYPE
  days = (mon,tue,wed,thu,fri,sat,sun);
  weekday = mon..fri;
  index = 0..75;

VAR  workday : weekday;
  array_index : index;
```

When you use a variable of subrange type in a DISPLAY command, Inspect displays the variable of the host type. If you want Inspect to display the ordinal representation of the variable, specify a numeric format; for example:

```
-PASOBJ-DISPLAY workday
WORKDAY = THU

-PASOBJ-DISPLAY workday IN DECIMAL
WORKDAY = 3
-PASOBJ-DISPLAY array_index
ARRAY_INDEX = 10
```

You can modify variables of subrange type using the host type of the variable or an ordinal value of the subrange; for example:

```
-PASOBJ-MODIFY workday = mon
-PASOBJ-MODIFY workday = 0
```

This example shows the internal attributes of the objects from the previous declarations:

```
-PASOBJ-INFO IDENTIFIER workday
WORKDAY: VARIABLE
storage^info:
TYPE=DEFINED TYPE, ELEMENT LEN=8 BITS, UNIT SIZE= 1 ELEMENTS
access^info:
#GLOBAL+1 WORDS
structure^info:
CHILD= WEEKDAY

-PASOBJ-INFO IDENTIFIER array_index
ARRAY_INDEX: VARIABLE
storage^info:
TYPE=DEFINED TYPE, ELEMENT LEN=16 BITS, UNIT SIZE= 1 ELEMENTS
access^info:
#GLOBAL+2 WORDS
structure^info:
CHILD= INDEX

-PASOBJ-INFO IDENTIFIER weekday
WEEKDAY: DEFINED TYPE
access^info:
TYPE= SUBRANGE, SUBRANGE CLASS= ENUMERATION, HOST ID= DAYS,
LOWER= MON, UPPER= FRI

-PASOBJ-INFO IDENTIFIER index
INDEX: DEFINED TYPE
access^info:
TYPE= SUBRANGE, SUBRANGE CLASS= INTEGER, LOWER= 0, UPPER= 75
```

Inspect Enhancements and Restrictions for Pascal

The following subsection discusses the difference between programming with Pascal and using Inspect to debug Pascal programs.

Length of Identifiers

Note that a Pascal identifier can be any length, but Inspect only accepts the first 31 characters.

Command Usage Guidelines for Pascal Programmers

The following guidelines for Pascal programmers using Inspect are arranged alphabetically by Inspect command name. Not all commands are listed.

BREAK

- You can set a code breakpoint at a line number, procedure name, or label name. If the number or name is not in the current scope, you must qualify it with a scope name.

  You can also set a code breakpoint at an offset from a line number or label, or at an offset from a primary entry point. Specify offsets as a number of statements or instructions.
- If you set a breakpoint at the entry point of a scope unit, the break event it triggers will occur before any initialization. Consequently, you should enter a STEP 1 S command after the break event.
- Because a data breakpoint is associated with a single word, an identifier designating multiple words (such as LONGINT) has only its first word marked as the breakpoint.

HELP

You can ask for help on the definitions of Inspect command parameters. Therefore, you can find out what Inspect recognizes as a Pascal data location, a Pascal code location, a Pascal expression, or a Pascal conditional expression.

INFO IDENTIFIER

- Like Pascal, Inspect requires complete name qualification of record elements. If you request a listing of the attributes of an unqualified data name, and the only instance of that name is in a record, Inspect will give you an error message. If the data name occurs in the scope twice—once as a simple variable and once as part of a record—requesting a listing of the attributes of the unqualified data name will produce an attribute listing for only the simple variable. You must qualify the name completely when you request a listing of the attributes for a record element.

- The data types reported by Inspect are not designated in terms of any one language. Therefore, Pascal integers are marked as BIN SIGN, while Pascal real, logical, and character items are marked as REAL, LOGICAL, and CHAR, respectively.

MODIFY

You can specify the name of a record in a MODIFY WHOLE command, but Inspect requires that you include the new values in the command. MODIFY does not prompt for values when you use the WHOLE clause.

SCOPE

If you have identifiers of the same name in different scope units, be sure that you qualify the identifiers enough for Inspect to distinguish them.

SET RADIX

Even if you set your input radix to hexadecimal, you must still prefix a hexadecimal value with a zero (or 16#) if its first digit is above nine; otherwise, Inspect interprets the value as an identifier.

13 Using Inspect With TAL and pTAL

Scope Units and Scope Paths

TAL and pTAL have only one type of scope unit: the procedure. When debugging a program written in TAL and pTAL, you must specify a procedure name whenever an Inspect command expects a scope unit.

Here is the syntax you use to identify TAL and pTAL scope paths in Inspect.

```
scope-path:
  #procedure
```

Code Locations

Here is the syntax you use to identify TAL and pTAL code locations in Inspect.

```
code-location:

   { scope-path                       } [ code-offset ]
   { [ scope-path. ] code-reference }

code-reference:  one of

   procedure
   subproc
 [ subproc. ] label
   [ subproc. ] entry-point
   statement-number
   #line-number
 [ (source-file) ]
```

```
code-offset:
   { + | - } num [ code-unit ]

code-unit:  one of
   INSTRUCTION[S]
   STATEMENT[S]
```

scope-path

specifies the procedure containing the code location. When followed by a code offset, *scope-path* specifies the base of the procedure; otherwise, *scope-path* specifies the primary entry point of the procedure.

[*scope-path*.] *code-reference*

specifies a named or numbered location in the procedure defined by the given scope path (or the current scope path if you omit the scope path).

procedure

specifies the primary entry point of the procedure. *procedure* must be the same as the procedure named in *scope-path* (or the current scope path).

subproc

specifies the entry point of a subprocedure.

[*subproc*.] *label*

specifies the statement following a given label in the source code. If the label is within a subprocedure, you must include *subproc*.

[*subproc*.] *entry-point*

specifies a secondary entry point. If the entry point is within a subprocedure, you must include *subproc*.

statement-number

specifies the statement beginning at the given statement number. To see statement numbers, use the SOURCE command after you have set your location format to statements. For more information, see SET LOCATION FORMAT on page 6-175.

#*line-number* [(*source-file*)]

specifies the statement beginning at a given line number in the source file.

(*source-file*) qualifies the line number by the source file containing it. You need to use this option only if the source code for the given procedure is in more than one file.

code-offset

specifies an offset from the code location defined by the preceding options. A positive offset (+) denotes code following the specified code location; a negative offset (-) denotes code preceding the specified code location. The amount to offset is specified by a given number of units. If you omit the unit specifier, Inspect selects INSTRUCTION as the code unit for TAL or pTAL. Inspect code units correspond to TAL and pTAL code units as follows:

INSTRUCTION	Specifies a machine-code instruction in the compiled TAL or pTAL program.
STATEMENT	Specifies a TAL or pTAL statement.

Usage Considerations

- Low-level code locations

  Low-level Inspect recognizes procedure names, but does not use any other symbol information. In low-level Inspect, therefore, you can only use code locations of the form:

  `#`*`scope-path`*` [ `*`code-offset`*` ]`

  This form represents an offset from the code base of a scope unit. Also, the code unit of *code-offset* in low-level Inspect is always INSTRUCTION.

- High-level code locations

  High-level Inspect recognizes procedure names as does low-level Inspect, but it also uses the symbol information created when you compile a scope unit with the SYMBOLS directive. Therefore, code locations in high-level Inspect can include label identifiers or line numbers.

- Specifying code locations by label

  You can use a TAL or pTAL label as the code reference in a code location. However, because Inspect also accepts scope units as code references, a conflict arises if a label's identifier is the same as the identifier for its containing scope unit. Inspect interprets the identifier as a reference to the scope unit, not to the label. Consequently, you must specify the code location of the label by its statement number, its line number, or its instruction offset.

- Specifying code locations by line number

  If no statement begins at the line number you specify, Inspect issues this warning:

```
** Inspect error 117 ** A subsequent line number is assumed: line-number
```

Inspect then uses the statement starting at the given line number. If more than one statement begins on the line you specify, Inspect uses the start of the first statement.

- A code location must include scope-name qualification if it refers to a location outside the current scope. Inspect assumes a code location is in the current scope if no explicit scope qualifier is stated.
- The STATEMENT code unit and TAL or pTAL statements

  Inspect recognizes these as statements:

  - Simple statements (including machine instructions in a CODE statement) that are not part of a composite statement
  - Statements in a compound statement (BEGIN...END)
  - The parts of a composite statement
- For TAL programs, use caution when specifying the procedure base as the location of a code breakpoint; doing so can result in the modification of P-relative data or alteration of subprocedure code.

Examples

The following examples assume a TAL procedure named TPROC that contains the label T^LABEL and the subprocedure TSUB, which in turn contains the label T^SUBLABEL:

Code Location	**Specifies**
#TPROC	The code base of the procedure TPROC.
#TPROC.TSUB	The entry point of subprocedure TSUB.
#TPROC.T^LABEL	The statement following the label T^LABEL.

These examples assume a current scope path of #TPROC:

Code Location	**Specifies**
TPROC	The primary entry point of the procedure TPROC.
TSUB.T^SUBLABEL	The statement following the label T^SUBLABEL.

Data Locations

Here is the syntax you use to identify TAL and pTAL data locations in Inspect.

```
data-location:
   [ scope-path [ (instance) ] . ]
   [[ scope-path. ] subproc [ (instance) ].  ] data-reference
   [ #data-block.                    ]
   [ ##GLOBAL.                       ]

instance:
   [ + | - ] integer

data-reference:  one of
   identifier
   data-reference '[' subscript-range ']'
   data-reference.identifier

subscript-range:
   expression  [ :expression ]
```

scope-path [(*instance*)]

specifies the procedure containing the data item.

(*instance*) identifies a specific activation of the data item's parent procedure. You need to specify an instance only when you want to identify a local data item in a recursive procedure.

[*scope-path*.] *subproc* [(*instance*)].

specifies the subprocedure containing the data item.

#data-block

specifies the data block containing the data item.

##GLOBAL

specifies the implicitly named global data block in a TAL or pTAL program.

data-reference

specifies the data item using TAL or pTAL syntax. The recursion in the definition of *data-reference* enables you to refer to complex TAL and pTAL data structures.

identifier

specifies a simple or pointer variable. When used in the DISPLAY command, *identifier* can also be the name of a structured variable; *identifier* then specifies the entire variable. When used in the INFO IDENTIFIER command, *identifier* can also be the name of a structured variable or user-defined data type; *`identifier`* then specifies the entire variable or the type definition.

`data-reference` `'['` *`subscript-range`* `']'`

specifies an array variable.

`subscript-range`

specifies the subscript of an array element or the subscript range of a group of array elements.

`data-reference.identifier`

specifies a field of a structure variable.

Default Values

If you do not specify scope-path, Inspect uses the current scope path.

Usage Considerations

- A data location must refer to an object defined in a procedure that was compiled with the SYMBOLS directive. The location can be anywhere in an active scope unit.
- You must qualify an identifier to denote a data item uniquely. If an unqualified identifier could refer to more than one data item, Inspect issues an error message stating that the reference is ambiguous (except when you are using the INFO IDENTIFIER command, which reports all occurrences of the identifier).
- To qualify a sublocal, you must precede the sublocal with it's subprocedure name.
- Inspect will not automatically expand a TAL identifier that was described by a TAL DEFINE declaration; however, the defined text is displayed when the identifier is specified in the INFO IDENTIFIER command.

Usage Considerations for TAL Programs

- Inspect recognizes the name associated with an index register by a USE statement as a data location identifier. You can display and modify the TAL index registers—R[5], R[6], and R[7]—by referring to the name associated with the index register.

  If you drop or push the name assigned to an index register, Inspect still allows access to the index register without displaying a warning.
- The values of sublocal variables and subprocedure arguments are accessible only if the subprocedure is current and if the stack pointer (S register) has not been modified by the subprocedure or by any subprocedure called by that subprocedure. The concept of instance does not apply to sublocal variables and subprocedure arguments.

- Within a scope, you can access the same index register with the USE and DROP statements more than once, as long you do not use the same name. In this example, the name INDEX^REG is assigned to a register more than once in a scope:

```
USE index^reg;
FOR index^reg := 1 TO 10 DO
   X[index^reg] := index^reg;
DROP index^reg;
...
USE index^reg;
```

If you attempt to display or modify INDEX^REG, Inspect displays this error message:

```
** Inspect error 98 ** Qualification required to resolve
ambiguous reference: identifier
```

You cannot further qualify the index register.

Examples

Given a TAL procedure named TPROC that contains INT T^INT and subprocedure TSUB, which in turn contains INT T^SUBINT, examples of data locations are:

Data Location	**Specifies**
#TPROC.T^INT	The most recent instance of T^INT.
#TPROC(-1).T^INT	The second-most recent instance of T^INT.
#TPROC(1).T^INT	The oldest (first) instance of T^INT.

This example assumes a current scope path of #TPROC:

Data Location	**Specifies**
TSUB.T^SUBINT	The current instance of T^SUBINT. Note that you cannot refer to instances of variables in subprocedures.

The example on the following page uses a stack TRACE to illustrate this behavior in subprocedures. Also notice that even though the subprocedure has parameters, Inspect cannot display them.

```
-C20FOBJ2-TRACE
Num Lang Location
     TAL   .SUB_SCOPE_N:  #SCOPE_N.#37(C20FTALK)
  0  TAL #SCOPE_N.#25(C20FTALL)
  1  TAL #C20FTAL2.#8(C20FTALP)
-C20FOBJ2-SOURCE
   #33
   #34              sub_scope_early:
   #35                    ! get current value of interval timer:
   #36                    !
  *#37                 call timestamp (time);
   #38                    !
   #39                    ! convert the time
   #40                    !
   #41              sub_scope_middle: call contime (date^time, time,
   #42                                                time[1], time[2]);
```

```
-C20FOBJ2-SOURCE BACK 40
   #1            !  ++++  subprocedure heading  ++++
   #2
   #3               subproc  sub_scope_n ( sub_scope_level, subparam_s8,
   #4                                      subparam_i16, subparam_i32,
   #5                                      subparam_f64, subparam_r32,
   #6                                      subparam_r64, subparam_s8p,
   #7                                      subparam_i16p, subparam_i32p,
   #8                                      subparam_f64p, subparam_r32p,
   #9                                      subparam_r64p );
   #10
-C20FOBJ2-TRACE ARG
Num Lang Location   (Arguments)
     TAL   .SUB_SCOPE_N:  #SCOPE_N.#37(C20FTALK)
  0  TAL #SCOPE_N.#25(C20FTALL)
SCOPE_LEVEL = 3, PARAM_S8 = ?6, PARAM_I16 = 20, PARAM_I32 = 72, PARAM_F64 =
272., PARAM_R32 =  2112., PARAM_R64 =  8320., PARAM_S8P = ?3, PARAM_I16P =
5,PARAM_I32P = 9, PARAM_F64P =  17., PARAM_R32P =  33., PARAM_R64P =  65.
  1  TAL #C20FTAL2.#8(C20FTALP)
-C20FOBJ2-LOG STOP
```

Expressions

Here is the syntax you use to create TAL and pTAL expressions in Inspect.

```
expression:
   condition [ { AND | OR } condition ]...

condition:
   [ NOT ] simple-exp [ rel-op simple-exp ]...

rel-op:  one of
    <     <=     =     >=     >     <>
   '<'   '<='   '='   '>='   '>'   '<>'

simple-exp:
   [ + | - ] term [ add-op term ]...

add-op:  one of
   +     -      '+'   '-'
   LOR   LAND   XOR

term:
   factor [ mult-op factor ]...

mult-op:  one of
   *    /    '*'    '/'   '\'
   <<   >>   '<<'   '>>'

factor:  one of
   primary   primary.<primary[:primary]>
```

```
primary:  one of
   data-location
   .data-location
   @data-location
   number
   (expression)
```

Usage Considerations

- Operator precedence is the same as the precedence defined for TAL and pTAL.
- To refer to the contents of the pointer named XX, use the expression (@XX). To refer to the contents of the indirect variable to which XX points, use XX.
- Inspect does not support the IF or CASE constructions in expressions; for example, these two Inspect commands are invalid:

```
-TALPGM-MODIFY x := IF y>5 THEN 12 ELSE 24
-TALPGM-MODIFY x := CASE r OF BEGIN y; y*y; y*y*y; OTHERWISE z; END
```

TAL and pTAL Data Types and Inspect

The following subsections discuss how Inspect handles and presents various TAL and pTAL data types.

Arrays

Inspect supports all TAL and pTAL array types, including arrays of structures. You can use a single element, a group of elements, or an entire array as the *data-reference* part of a data location:

Data Reference	Example	Where Valid
Single element	MYARRAY[5]	In any Inspect command
Range of elements	MYARRAY[1:7]	In any Inspect command except BREAK
Entire array	MYARRAY	In the INFO IDENTIFIER and DISPLAY commands only

Examples

This example assumes the TAL declaration:

```
INT d^array [0:9]
```

Here are the attributes of the array:

```
-TALOBJ-INFO IDENTIFIER d^array
D^ARRAY: VARIABLE
storage^info:
TYPE=BIN SIGN, ELEMENT LEN=16 BITS, UNIT SIZE=1 ELEMENTS
```

```
access^info:
'L' + 1  WORDS
dimension^info:
[0:9]
```

Structures and Substructures

Inspect supports TAL and pTAL structures and substructures, as shown in the following example, which assume these TAL declarations:

```
STRUCT alphabetics;
  BEGIN
    STRING caps [1:26];
    STRING smalls [1:26];
    STRING misc [1:10];
  END;

STRUCT name^middle^last;
  BEGIN
    STRING first^initial;
    STRING middle^name [0:19];
    STRING last^name [0:19];
    STRUCT birth;
      BEGIN
        INT year;
        INT month;
        INT day;
      END;
  END;
```

Here are the internal attributes of the structure ALPHABETICS:

```
-TALOBJ-INFO IDENTIFIER alphabetics
ALPHABETICS: VARIABLE
storage^info:
TYPE=WORD STRUCT, ELEMENT LEN=16 BITS, UNIT SIZE=31 ELEMENTS
access^info:
'L' + %136  WORDS
structure^info:
CHILD=CAPS
```

Here are the internal attributes of one of the fields in the structure ALPHABETICS:

```
-TALOBJ-INFO IDENTIFIER alphabetics.smalls
SMALLS: VARIABLE
storage^info:
TYPE=CHAR, ELEMENT LEN=8 BITS, UNIT SIZE=1 ELEMENTS
access^info:
(parent) + 0 + %15 WORDS
dimension^info:
[1:26]
structure^info:
PARENT=ALPHABETICS,SIBLING=MISC
```

This example displays the structure NAME^MIDDLE^LAST:

```
-TALOBJ-DISPLAY name^middle^last
NAME^MIDDLE^LAST =
  FIRST^INITIAL = "M"
  MIDDLE^NAME[0] = "Treuhardt          " ?0
  LAST^NAME[0] = "Korhummel          " ?0
  BIRTH =
    YEAR = 1844
    MONTH = 10
    DAY = 31
```

This example displays the structure NAME^MIDDLE^LAST, suppressing the identifiers and quotes:

```
-TALOBJ-DISPLAY name^middle^last PLAIN
M
Treuhardt           ?0
Korhummel           ?0
  1844
  10
  31
```

This example displays a field of the structure NAME^MIDDLE^LAST:

```
-TALOBJ-DISPLAY name^middle^last.last^name
NAME^MIDDLE^LAST.LAST^NAME[0] = "Korhummel          " ?0
```

This example displays the substructure BIRTH. Note that you must specify its parent structure, NAME^MIDDLE^LAST:

```
-TALOBJ-DISPLAY name^middle^last.birth
NAME^MIDDLE^LAST.BIRTH =
  YEAR = 1844
  MONTH = 10
  DAY = 31
```

This example displays a field of the substructure BIRTH:

```
-TALOBJ-DISPLAY name^middle^last.birth.year
NAME^MIDDLE^LAST.BIRTH.YEAR = 1844
```

Command Usage Guidelines for TAL and pTAL Programmers

The following guidelines for TAL and pTAL programmers using Inspect are arranged alphabetically by Inspect command name. Not all commands are listed.

BREAK

- If you set a breakpoint at the entry point of a scope unit, the break event it triggers will occur before any initialization. Consequently, you should enter a STEPþ1S command after the break event.

- When you use a code offset to specify the location of a code breakpoint, Inspect does not ensure that the location specifies executable code. If the location is in a P-relative array, the breakpoint may alter the data in the array.
- Because a data breakpoint is associated with a single word, an identifier designating multiple words (STRUCT, INT(32), REAL(64) and so on) has only its first word marked as the breakpoint. If an identifier is declared to be at an address within such a multiple-word item, allowing you to read or modify a subsequent word of such a variable without affecting the primary word, the debugging facility will not signal a debug event.
- If you set a data breakpoint within a string item, and your program uses a move statement (X ':=' Y FOR 25) to move a value to the string, the break event will occur before the move operation is complete.

DISPLAY

To display the contents of a TAL or pTAL pointer (that is, the address to which it points), use an address-of expression; for example:

```
-TALOBJ-DISPLAY ( @pointer )
```

HELP

You can ask for help on the definitions of Inspect command parameters. Therefore, you can find out what Inspect recognizes as a TAL or pTAL data location, a TAL or pTAL code location, a TAL or pTAL expression, or a TAL or pTAL conditional expression.

INFO IDENTIFIER

- If you request a listing of the attribute of an unqualified data name, Inspect will list all possible instances of that data name (that is, the same name in various subprocedures or substructures).
- If you request a listing of the attributes of an entire procedure, they are listed in an order determined by the compiler. For TAL and pTAL, this order is:

  1. Data identifiers that are not structure names
  2. Structure names
  3. Labels

- In addition to providing attribute information for code and data locations, Inspect can provide information about TAL and pTAL defines. When you request the attributes of a define, Inspect displays the first 68 characters of the text associated with the define.

MODIFY

- To modify a string data item, use the subscript range form:

  ```
  -TALOBJ-MODIFY x^string [0:10] := "abcdefghij"
  ```

- You can specify the name of a structure in a MODIFY WHOLE command, but Inspect requires that you include the new values in the command. MODIFY does not prompt for values when you use the WHOLE clause.
- As in TAL and pTAL, you can MODIFY a variable that is equated to another variable.
- As in TAL and pTAL, you can apply a subscript to a data item that is not declared to be an array.

SCOPE

- In TAL, the scope of an identifier is always the name of the procedure (not subprocedure) or data block in which the identifier is declared. TAL and pTAL provide the ##GLOBAL data block for all global variables that are not declared in a named block or a private block.
- If you have identifiers of the same name in different scope units, be sure that you qualify the identifiers enough for Inspect to distinguish them.

SET RADIX

Even if you set your input radix to hexadecimal, you must still prefix a hexadecimal value with a zero (or %H) if its first digit is above nine; otherwise, Inspect interprets the value as an identifier.

STEP

- The STEP command defaults to STATEMENTS if no code-unit is specified.
- The STEP command requires caution if CASE statements or FOR loops are in the path. The stepping behavior of these two statements is unexpected.

  Remember that a CASE selects one statement from a set of statements, depending on the value of a numeric expression. A STEP of one statement from the beginning of a CASE statement takes you to the end of the entire CASE statement. A subsequent STEP of one statement will take you to the selected case or default statement.

  If your process is at the beginning of a single-statement FOR loop (the loop body is a single statement, not a block), entering STEPþ1S gets you to the beginning of the single statement, and entering STEPþ2S completes execution of the loop.

- By default the STEP command steps over procedure and subprocedure calls. In IN option can be used to step into calls made to procedures/subprocedures that are contained in the user code or the user library.

If you are stepping over a procedure and a debugging event occurs while that procedure/subprocedure is executing or a procedure/subprocedure that it calls is executing, execution is suspended where the event occurs. Among other reasons, this can happen if you placed a breakpoint in a called procedure/subprocedure. You can use the STEP OUT command to return back to the level at which execution was originally stepped.

TRACE

The TRACE ARGUMENTS command shows the arguments for the primary entry point of a scope unit. If a secondary entry point has a different set of arguments, TRACE ARGUMENTS does not display them.

14 Using Inspect in an OSS Environment

Starting an Inspect Session

You can start an Inspect session in OSS in any of these four ways:

- From the TACL prompt, enter:

  ```
  >RUN INSPECT or INSPECT
  ```

  From the OSS shell, enter:

  ```
  gtacl -p inspect
  ```

- From TACL prompt, you enter:

  RUND *program*

  From the OSS shell, you enter:

  run -debug -inspect=on *program*

- Issue a debug request on a running process.
- A debug event occurs in a program.

In all cases, Inspect will be started as a Guardian process.

Inspect looks for an EDIT file named INSPLOCL in the volume and subvolume containing the Inspect program file. After reading the INSPLOCL file, the Inspect process looks for an EDIT file named INSPCSTM in the logon volume and subvolume of the creator of the process being debugged. All OSS users have a corresponding Guardian logon volume and subvolume.

To avoid incompatibilities between Guardian and the OSS terminal I/O when debugging an OSS process, change Inspect's command terminal to an existing Guardian terminal using the TERM command as soon as Inspect comes up.

Ending an Inspect Session

Ending an Inspect session is the same regardless of the platform or environment. An Inspect session can be stopped in the following ways:

- You exit by using the EXIT command or the RESUME * EXIT command.
- Your last program has completed.
- Your program is stopped by someone else.

Inspect's System Type

Inspect uses the systype to determine certain command option defaults and to determine the type of the file names accepted. A program can have a Guardian or an OSS systype, Inspect's systype will change to the systype of the current program.

Inspect commands accept different syntax depending on the current systype. For example, if the current program is an OSS process, but Inspect's systype is Guardian, the following command results in a syntax error because the file specified is an OSS pathname and not a Guardian file name.

```
-PROG- SOURCE AT #200 FILE /usr/src.file.c
```

A program's systype is determined by the type of process it is (Guardian for Guardian processes, or OSS for OSS processes). Commands have been added to Inspect to change and display the current systype. For more information, see SET SYSTYPE on page 6-192 and SELECT SYSTYPE on page 6-170.

The main result of the systype setting is the type of file name Inspect accepts. If the current systype is Guardian, Inspect reads all file names as Guardian file names. Inspect's systype resets at each event. Inspect's default systype is Guardian.

File Name Resolution

Inspect maintains a default subvolume and a default OSS directory. The current default subvolume is used to resolve file names when Inspect's systype is Guardian. The current default OSS directory is used to resolve pathnames when Inspect has an OSS systype.

If Inspect is started using the command interpreter (either TACL or at the OSS shell), the default subvolume and default OSS directory are the command interpreter defaults, that is, current working subvolume or current working directory. In all other cases, such as RUND from TACL or `run -debug` from the OSS shell, the default subvolume and default OSS directory is the logon defaults of the creator of the program being debugged.

These commands can be used to resolve file name inconsistencies:

- The VOLUME command can be used to change the default subvolume.

- The CD command can be used to change the default OSS directory.
- The ENV command can be used to display both the defaults.

For more information, see VOLUME on page 6-223, CD on page 6-27, or ENV on page 6-81.

Save Files

The same events that cause a save file to be created for Guardian processes cause save files to be created for OSS processes. That is, if the process ABENDs or otherwise terminates abnormally, or you type Inspect's SAVE command to create a save file explicitly, a save file will be created.

If your process terminates abnormally, save files are only created if the SAVEABEND flag is on that process, as opposed to Inspect's SAVE command which creates a save file regardless of the value of the SAVEABEND flag. In addition, a save file will only be created if the process is being actively debugged, or the parent process has sufficient access to the debug process.

Process save files for OSS processes are created in the current working directory of the ABENDing process. The files are owned by the effective User ID (UID) of the process requesting the save file (in the case of ABEND, the process itself is requesting the file). Save files are created with permission bits allowing read and write access to owner, read access to group, and read access to other. Process save files created automatically by a process ABENDing will be named ZZSA*nnnn*, where *nnnn* is a four digit number.

Signals

Two complimentary Inspect commands support OSS signals, INFO SIGNALS and MODIFY SIGNALS. INFO SIGNALS displays signal information for the current program and MODIFY SIGNALS changes signal handlers for OSS processes which are the current program. Use the commands HELP INFO SIGNALS and HELP MODIFY SIGNALS for an in-depth description of signals.

Signals allow two processes to communicate with each other. Two complimentary Inspect commands support OSS signals, INFO SIGNALS and MODIFY SIGNALS. INFO SIGNALS displays signal information for the current program and MODIFY SIGNALS changes signal handlers for OSS processes which are the current program. Refer to the *Open System Services Programmer's Guide* for an in-depth description of signals.

Considerations

- You cannot set breakpoints on signal events, but you can set a breakpoint on a signal handler.
- Inspect cannot be used to send signals to other processes.

Examples

1. This example illustrates the MODIFY command with default arguments.

```
-test4--- Default arguments for MODIFY SIGNALS
-test4---
-test4-MODIFY SIGNAL SIGUSR1
Signal SIGUSR1(16)
Handler = SIG_DFL := #func_1
Mask = 0 := 100
Flags = 0 := 500
-test4-INFO SIGNAL SIGUSR1
Signal          Handler             Mask          Flags
SIGUSR1(16)    func_1             100           500
```

2. This example illustrates MODIFY without default arguments.

```
-test4-INFO SIGNAL SIGUSR1 DETAIL

                             Signal: SIGUSR1(16)
                               Handler: func_1
                                  Mask: 100
                                 Flags: 500
-test4-SIGNAL SIGUSR2 := #func_2, 150, 700
-test4-INFO SIGNAL SIGUSR2 DETAIL

                             Signal: SIGUSR2(17)
                               Handler: func_2
                                  Mask: 150
                                 Flags: 700
```

Source Files

Inspect displays source from OSS files. OSS files consist of ASCII text with lines terminated with a (new-line) ASCII 10 character. Because line numbers automatically increment for OSS files, but not for Guardian files, lines numbers for corresponding converted source files may differ. If your file was originally compiled in an OSS environment, the lines numbers will correspond.

Usage Guidelines

This section presents guidelines for OSS programmers using Inspect. Table 14-1 maps DBX commands to equivalent Inspect commands. “N/A” indicates that no corresponding command exists.

Table 14-1. DBX/Inspect Command Map

DBX	Inspect	DBX	Inspect
/	SOURCE SEARCH	quit	STOP/EXIT
?	SOURCE SEARCH	record input	LOG with the INPUT option
address	DISPLAY, ICODE	record output	LOG with the OUTPUT option

Table 14-1. DBX/Inspect Command Map

DBX	Inspect	DBX	Inspect
alias	ADD ALIAS, LIST ALIAS	return	STEP with the OUT option
assign	MODIFY	run	RESUME
catch	N/A	rerun	N/A
cont	RESUME	set	N./A
conti	N/A	sh	N/A
delete	DELETE, CLEAR	source	OBEY
down	SCOPE	status	LIST BREAKPOINT
dump	INFO SCOPE, TRACE command with ARGUMENTS option	step	STEP with the IN option
edit	N/A	stepi	STEP with the ICODE option
file	N/A	stop	BREAK
func	SCOPE	stopi	BREAK
goto	RESUME AT	trace	BREAK with the THEN clause
help	HELP	tracei	N/A
history	HISTORY	unalias	DELETE ALIAS
ignore	MODIFY with the SIGNAL option	unset	N/A
list	SOURCE	up	SCOPE
next	STEP	use	SOURCE ASSIGN
nexti	STEP with the ICODE option	whatis	INFO IDENTIFIER
playback input	OBEY	when	BREAK with the.IF...THEN clauses
playback output	N/A	where	TRACE
print	DISPLAY	whereis	MATCH with the IDENTIFIER option, MATCH with the SCOPE option
printregs	DISPLAY with the REGISTER option	which	N/A

Table 14-2. Inspect Commands Without a DBX Counterpart

Command	Description
Info Segment	List currently allocated extended segments
Info Opens	List files opened by current process
Add Program	Add a program to the current Inspect session
List Program	Inspect is able to debug multiple programs
Select Program	Select the current program. This can also be used to load new symbol files for the specified program.
Display	Inspect’s Display command has a richer set of options and arguments than DBX’s print command
Info Object	Displays information about the current processes object files

15 Using Inspect on a TNS/R System

If you want to use Inspect on a TNS/R system, you should read this section as an introduction, in addition to one of the following sections, as appropriate for your debugging target (TNS/R native code or TNS accelerated code):

Section 16, Using Inspect With Accelerated Programs on TNS/R Systems

Section 17, Using Inspect With TNS/R Native Programs

TNS/R Overview

TNS/R computers support the HP NonStop operating system and existing applications, but are based on reduced instruction set computing (RISC) technology. Inspect supports the debugging of TNS, accelerated, and native TNS/R programs on TNS/R systems.

Much of the code in HP-supplied software products for TNS/R systems has been produced by TNS/R native compilers. You can also use native compilers to produce your own native TNS/R code. (Refer to the *C/C++ Programmer's Guide*, the *COBOL Manual for TNS and TNS/R Programs*, and the *pTAL Reference Manual*.) TNS/R native code consists of RISC instructions that have been optimized to take advantage of the RISC architecture. Program files containing such code are called native program files.

RISC technology uses hardware and software components to increase program performance. Hardware features which result in increased program performance include the pipelining of instructions, which allows multiple instructions to be executed in parallel, and an increased number of general purpose registers. Native TNS/R programs require no additional optimizations to achieve optimum program performance.

TNS programs, produced by TNS compilers rather than TNS/R native compilers, also execute on TNS/R systems. TNS programs contain TNS object code. Program files containing TNS object code are called TNS program files. (Even though TNS processors are no longer supported, programs executed on TNS processors can execute on TNS/R processors with very few exceptions. These exceptions are described in the *Accelerator Manual* and the *Guardian Application Conversion Guide.*)

For some TNS program files, you can significantly improve performance by processing them with the Axcel accelerator to make use of performance features of the RISC instruction set. The accelerator processes a standard TNS object file and augments that file by adding the equivalent RISC instructions. TNS object files that have been optimized by the accelerator are called accelerated object files, or accelerated program files if they include a main procedure. Running accelerated program files can significantly improve performance over simply running TNS program files directly on the TNS/R processor. The accelerator, however, provides optimization options that can affect program debugging. For more information, see Section 16, Using Inspect With Accelerated Programs on TNS/R Systems.

To make debugging easier than with most RISC-based systems, Inspect determines the consistency of the program state, classifying synchronization points as either memory-exact or register-exact points. These points give you information about your program, such as whether memory and registers are up to date.

Executing Non-Accelerated Programs

You can execute TNS object files on a TNS/R machine without any change. Use of the Axcel accelerator is only necessary when program performance is an issue.

When you execute a TNS object file on a TNS/R machine, the TNS instructions that it contains are executed by means of millicode. Millicode implements the actions of TNS machine instructions using the appropriate TNS/R instructions. It may be easier to debug a program executing TNS object code than to debug a program executing code generated by the accelerator.

When TNS instructions are executed on a TNS/R system, the TNS machine state (the register stack, S, P, E, and L registers) is maintained as if you were executing your program on a TNS machine. Because the TNS machine state is maintained, no change is required to run existing TNS programs on TNS/R systems and, with very few exceptions, debugging is the same.

Figure 15-1 illustrates the methods of executing TNS programs on a TNS/R system.

On a TNS/R machine, Inspect supports the debugging of two types of programs:

- Non-accelerated programs, programs that are executed as is.
- Accelerated programs, programs that have been optimized by the accelerator.

Three mechanisms maintain the TNS machine state sufficiently to ensure correct program behavior, but they differ in the extent to which the machine state is maintained. The first, TNS object code execution, most faithfully maintains the TNS machine state—for instance, all TNS instruction side effects, such as changes to ENV register flags, even if not used by the program code, are maintained.

Executing Accelerated Programs

The Axcel accelerator can be used to optimize program performance. The accelerator generates optimized TNS/R code, which is written to an object file that is a superset of the original object file. The TNS/R instructions contained in this object file can be directly executed by the TNS/R machine. The accelerator produces programs that execute significantly faster, but may be more difficult to debug.

The accelerator takes as input an executable TNS object file, and produces as output a file containing both that original code and logically equivalent optimized TNS/R (RISC) instructions. The accelerator produces for each TNS instruction its functional equivalent for the TNS/R system, in the form of either:

- A sequence of TNS/R instructions
- A call to a millicode routine

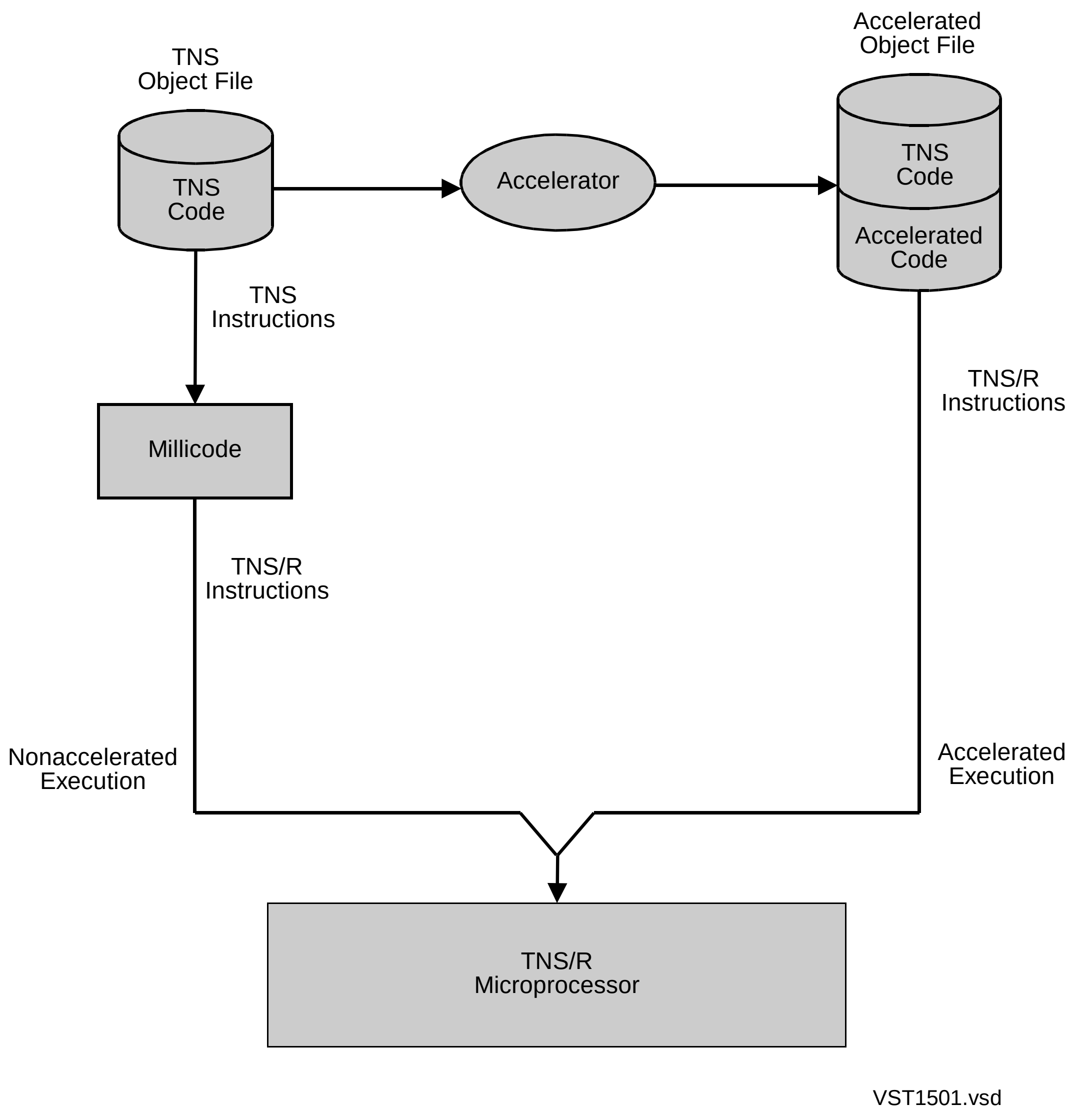
Sequences of TNS/R instructions that correspond to individual TNS instructions may be interleaved as a result of optimizations.

The accuracy with which TNS/R code maintains the TNS machine state varies. Based on its analysis of the program, the accelerator is able to generate code that maintains only the portions of the TNS machine state that are required by the program. The optimization level and options used with the Axcel accelerator also have some effect on the extent to which the TNS machine state is maintained. For more information, see Section 16, Using Inspect With Accelerated Programs on TNS/R Systems.

Some TNS program constructs require evaluation at execution time, and their behavior in all cases cannot be predicted by the accelerator. For such constructs, such as a SETP instruction, the accelerator generates code that checks if the conditions required by the generated code are met. If so, execution of TNS/R instructions continues; if not, execution transfers to the corresponding TNS instructions, which are executed until a point is reached where the execution of TNS/R instructions can resume.

With programs that use user libraries, one component may be accelerated while the other component is not. For example, you can call an accelerated user library from a program that has not been accelerated. Similarly, it is possible to call a user library that has not been accelerated from a program that has been accelerated.

Figure 15-1. TNS Program Execution on a TNS/R System



General TNS/R Debugging Considerations

These considerations apply when debugging non-accelerated, accelerated, and native programs on a TNS/R machine.

- Data breakpoints

  Data breakpoints might be reported at different locations than on a TNS system.

- Data representation

  Data is represented the same as on TNS systems. Data pointer values and stack frames are the same. The one exception is 32-bit code pointers, such as extended pointers to P-relative arrays, which might have different values on a TNS/R system due to differences in the extended addressing of code. References through such pointers will yield the same results as on TNS systems.

- System Global Space

  TNS/R systems differ in how they handle nonprivileged references to the System Global (SG) space. TNS systems redirect such references to the user data space, whereas a trap is reported on a TNS/R machine.

  When performing nonprivileged debugging on a TNS/R processor, Inspect reports an error if a System Global address is specified.

- Address Wrap

  The TNS and TNS/R systems differ in how they handle data stack addresses that overflow 16 bits. TNS systems discard the high-order bits, in effect wrapping the address to the beginning of the data stack. A trap is reported if this occurs on a TNS/R machine.

Debugging Non-Accelerated Programs

Debugging non-accelerated programs on a TNS/R processor is nearly identical to debugging them on a TNS system. The only difference is in the area of data breakpoints, which may be reported at slightly different locations than on a TNS machine. You can debug non-accelerated programs both at the source level and at the TNS machine level.

Debugging Accelerated Programs

When debugging accelerated programs, there are minor restrictions when you debug at the source level and significant restrictions when you debug at the TNS machine level. Limited debugging is available at the TNS/R machine level.

Inspect supports the existing TNS model of debugging for accelerated programs, subject to certain constraints. Source-level debugging is least affected by these constraints, while TNS machine-level debugging is significantly limited. This TNS view of an accelerated program may be inconsistent at some locations because:

- The contents of memory might be stale, that is, data may be stored in registers and will be stored in memory later.
- Data may be pre-fetched from memory into registers, causing the modification of memory to have no effect on program behavior.
- The TNS registers may not be up to date.

The run-time differences between running programs on TNS and TNS/R systems, some of which can have an important effect on debugging, are described in the *Accelerator Manual*. As that manual notes, programs executed on the TNS/R systems might fail differently than when executed on a TNS system; for example, traps can be detected at different locations.

Recommendations

To simplify debugging:

- Complete application development and debugging before using the accelerator.
- Optimize a version of the program containing symbols with the accelerator, using the default debugging option, ProcDebug, and perform final testing.
- Examine your program for variances if there are differences in program behavior.

Debugging Accelerated Programs at the Source Level

Debugging accelerated programs at the source level is very similar to debugging TNS programs at the source level, with these exceptions:

- Data breakpoints, also referred to as memory-access breakpoints, may be reported at different locations.
- Some statement boundaries may have been deleted (for example, code for one statement may be combined with code for an adjacent statement).
- Stepping may leave a program at a different location when statements have been deleted and for some loop constructs.
- The locations to which execution can be arbitrarily resumed are limited.
- At some statements, memory modification could have no effect.
- After data breakpoints, displayed memory may be out of date.

Debugging Accelerated Programs at the TNS Machine Level

When debugging accelerated programs at the TNS machine level, you will notice these significant differences and limitations:

- Code breakpoints cannot be set at most instructions that are not at the beginning of statements.

- Displayed TNS register values are often out of date.
- You can rarely modify TNS registers to affect program behavior.

Debugging Accelerated Programs at the TNS/R Machine Level

Although it is rarely necessary to debug accelerated programs at the TNS/R machine level, Inspect provides a limited set of features which allow you to:

- Display instructions, including the correspondence between TNS and TNS/R instructions
- Display TNS/R registers
- Modify TNS/R registers

The SELECT DEBUGGER DEBUG command can be used to access other TNS/R debugging functionalities.

Debugging TNS/R Native Programs

Debugging a TNS/R native program is similar to debugging the RISC portions of an accelerated program, but there are a few differences.

- In TNS/R native mode, local variables are sometimes cached in registers. Attempting to modify a local variable or use it for setting a memory access breakpoint, for example, can have unexpected results.
- In highly optimized native object code, parameter values are sometimes cached in registers, making their exact location unpredictable.
- In TNS/R native mode, unlike accelerated mode, there are no TNS instructions corresponding to the RISC instructions.

Performance and Debugging of TNS/R Programs

The following subsections detail how RISC processors improve program performance and how specific performance elements may affect your code.

Register Usage

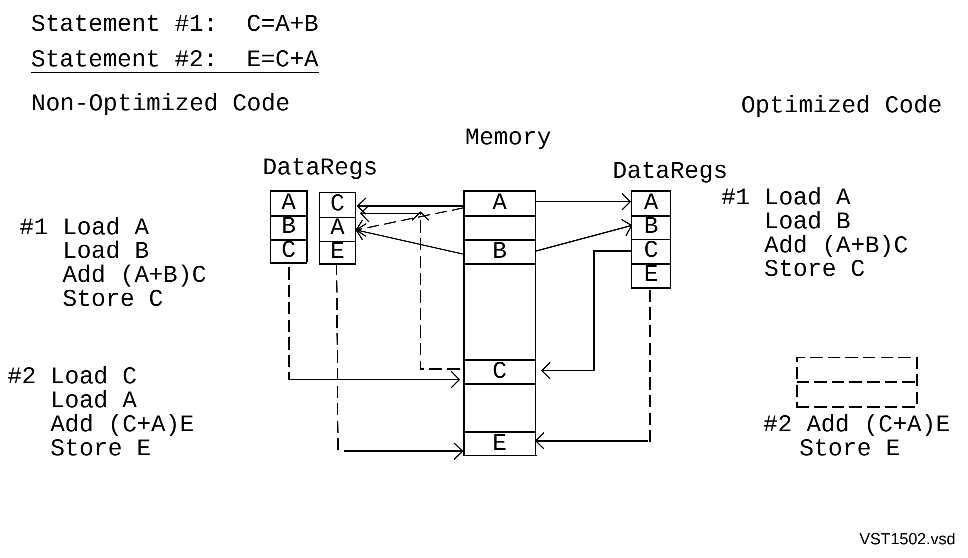
The TNS/R machine has 32 general purpose registers that can be used to store values and intermediate results of computations. It is much faster to access values stored in registers than to fetch them from memory. Therefore, an optimization strategy is used to generate code such that frequently used values are available in registers. Storing frequently used values minimizes the number of memory loads and stores that are performed while also ensuring that locations exist, usually at the beginning of statements, where the TNS program state is consistent.

Example

This example illustrates how optimized code increases performance by utilizing registers to reduce memory accesses.

The use of general purpose registers by the accelerator can have a number of effects on debugging. It may change the memory reference patterns of the program, causing some data breakpoints, particularly read breakpoints, not to be triggered. It also results in locations at which displayed memory may not be accurate (if a more recent value is stored in a register) and locations at which memory cannot be “safely” modified (a more recent copy may be stored in registers).

Figure 15-2. Memory Access by Optimized vs. Non-Optimized Code



Pipeline Instruction Processing

Like other RISC processors, TNS/R processors pipeline instruction execution, allowing components of instructions to be decoded and executed in parallel. This allows the execution of most instructions to complete in one cycle. The exceptions are load and branch instructions, which require an additional cycle.

The effect that pipelining has on debugging is that it results in a four cycle delay before data breakpoints are reported.

Optimizations With Loads

Load instructions load values from memory. On TNS/R processors, load instructions do not complete in the single cycle required to complete most instructions.

The position following a load instruction is referred to as a load delay slot. Dummy instructions or NOP instructions can always be placed in delay slots. As part of its optimization process, the compiler attempts to place useful instructions in load delay slots. The only requirement is that the instruction cannot depend on the data being loaded. Filling delay slots with useful instructions changes the order of instructions but yields improved program performance. Reordering of instructions usually has little effect on source-level debugging, but it may affect machine-level debugging.

Optimizations With Branches

Branch instructions control the path of instructions executed by a program. On TNS/R processors, branch instructions require an extra machine cycle to complete. The position following a branch instruction is referred to as a branch delay slot; it contains an instruction that will be executed before the branch occurs.

One of the optimizations performed by the accelerator is the filling of such branch delay slots with useful instructions; otherwise, delay slots are filled with NOP instructions. Filling delay slots often results in the instruction sequence being reordered, yielding improved program performance. Branch optimizations can result in statements being merged, thereby reducing the number of breakpoint locations.

Another optimization performed involves branch instructions is the elimination of “branch chains.” In instances where a branch instruction branches to another branch instruction, a single branch instruction may be generated that branches to the target of the second instruction.

Example

This example illustrates how the accelerator fills load and branch delay slots with useful instructions.

```
    #22            i := j;
    #23            IF j = 4 THEN
```

This example sows the TNS and TNS/R instructions for the preceding source code:

```
#22
  %000006:  > LOAD  L+006         %h70420030:  > LH    t5,12(fp)
  %000007:    STOR  L+005         %h70420034:    LI    at,4
  %000010:    LOAD  L+006         %h70420038:    BNE   t5,at,0x70420088
  %000011:    CMPI   +004         %h7042003C:    SH    t5,10(fp)
  %000012:    BNEQ   +010
```

The accelerator has placed the LI instruction in the delay slot of the LH instruction and the SH instruction in the delay slot of the BNE instruction. This results in the following operations being performed:

1. Load the value of j
2. Load the value 4
3. Branch if j <> 4

4. Store the value of j in i

TNS Instruction Side Effects

Many TNS instructions update the machine state, such as registers R0-R7 and status flags in the TNS environment register, the most common being the condition code flags. The effects that instructions have on environment register flags are often referred to as instruction side effects.

The TNS/R processor does not have the same registers as the TNS processor. Therefore, when the accelerator translates a program, it must generate code that updates the TNS execution model to reflect instruction side effects used by the program. Since programs often do not make use of instruction side effects, the accelerator does not generate code for side effects that are not used.

This has little impact on source-level debugging. When debugging at the TNS machine level, it could result in the displayed TNS registers having different values than they would on a TNS machine. For example, the condition code flags are only valid if they are used by the program.

Debugging Programs at the TNS/R Machine Level

Inspect provides some functionality for examining the TNS/R machine state when debugging either TNS/R native or accelerated programs. This includes the ability to display and modify TNS/R machine registers and to display TNS/R instructions. A complete set of TNS/R machine-level debugging functionalities, including the ability to set breakpoints on TNS/R instructions, is available by invoking Debug from within Inspect.

Note. Debugging at the TNS/R machine level is rarely necessary

What You Need to Know

To debug at the TNS/R machine level, you must have knowledge of the TNS/R architecture, including the instruction set, registers usage, addressing, transitions from TNS/R to TNS code, and variances between TNS and TNS/R systems.

- For more information about TNS/R instruction set, see *MIPS RISC Architecture*, by Gerry Kane, Prentice Hall: 1989.
- For more information about the accelerator, transitions into TNS code and variances between TNS and TNS/R systems, see the *Accelerator Manual.*
- For more information about register usage and addressing, see the *NonStop S-Series Server Description Manual*.
- For more information about Debug, see the *Debug Manual.*

- For more information about compilers, see the *C/C++ Programmer's Guide*, the *COBOL Manual for TNS and TNS/R Programs*, and the *pTAL Reference Manual*.

TNS/R Breakpoints

When debugging Accelerated programs, you can list and clear TNS/R breakpoints, but not set them. You must set TNS/R breakpoints from Debug. There are specific rules that apply to setting TNS/R breakpoints; for more information, see the *Debug Manual.* If you set a TNS/R breakpoint and the program transfers to execute TNS code (as opposed to TNS/R instructions) at that point in the program, the TNS/R breakpoint will not be triggered.

When debugging TNS/R native programs, you can set breakpoints at any location in the program. To set a breakpoint in native code, you can either specify a scope name or a machine-level code address.

Example

This example shows using Debug to set a TNS/R breakpoint and clearing TNS/R breakpoints using the CLEAR command with the appropriate breakpoint ordinal:

```
-PTALIN-SELECT DEBUGGER DEBUG
DEBUG 000061, 000207, UC.00
244,00,083-bn 704201d0
        N: 7042.01D0      INS: SH    t5,-4(sp)
244,00,083-INSPECT
INSPECT
244,00,083  PTALIN  #M.#75.1(TALIN)
-PTALIN-LIST BREAKPOINT
Num Type Subtype Location
  1 Code DEBUG   TNS/R %h704201D0
-PTALIN-RESUME
INSPECT  TNS/R BREAKPOINT 1: TNS/R %h704201D0
244,00,083  PTALIN  #M.#78(TALIN)
 ** Inspect warning 359 **** Current location is not a memory-exact point;
                       displayed values may be out of date;
                       the location reported is an approximate TNS
location
-PTALIN-CLEAR 1
Breakpoint cleared: 1  Code DEBUG TNS/R %h704201D0
-PTALIN-LIST BREAKPOINT
 ** Inspect warning 202 **** No breakpoints exist
```

TNS/R Machine Registers

Within Inspect, you can display the value of these TNS/R machine registers when debugging TNS/R programs:

```
tns/r-register: one of

    $0   $1   $2   $3...$31
    $HI  $LO
    $PC
    tns/r-register-alias
```

Register aliases are used when TNS/R instructions are displayed. General purpose TNS/R registers may also be identified using one or more aliases.

These mapping of aliases to registers is supported by Inspect:

Register	Alias	Register	Alias
$1	$AT	$17	$S1
$2	$V0	$18	$S2
$3	$V1	$19	$S3
$4	$A0	$20	$S4
$5	$A1	$21	$S5
$6	$A2	$22	$S6
$7	$A3	$23	$S7
$8	$T0	$24	$T8
$9	$T1	$25	$T9
$10	$T2	$26	$K0
$11	$T3	$27	$K1
$12	$T4	$28	$GP
$13	$T5	$29	$SP
$14	$T6	$30	$FP
$15	$T7	$31	$RA
$16	$S0		

Machine Code Addresses

Several of the commands described in the following sections accept machine code addresses. When debugging accelerated programs, TNS/R code addresses must be preceded by the TNS/R clause to be distinguished from TNS addresses. (The TNS/R clause can also be abbreviated as “R”.

TNS/R code addresses can be specified as any expression that computes an integer result that can be expressed in 32 bits. Examples of TNS/R address expressions as used with the ICODE command include:

```
ICODE R %h70421164
ICODE R %h70421164  - 3 * 4        (1)
ICODE R  $pc - 2 * 4 for 3         (2)
```

1. This lists the instruction three instructions prior to the specified address (instructions are four bytes long).
2. This lists three instructions starting two instructions prior to the current instruction.

Usage Considerations

- The current input radix is used to evaluate all numbers unless a different base is explicitly specified (using one of the standard base prefixes). It is most useful to use hexadecimal when inputting TNS/R addresses.
- Register names can be used in expressions.

Save Files

TNS/R register values are stored in save files created for accelerated and native programs.

TNS/R Machine-Level Commands

Inspect provides minimal support for machine-level debugging of native and accelerated programs. The following subsections briefly describe the commands Inspect supports. For more information, see Section 6, High-Level Inspect Commands.

DISPLAY REGISTER

You can use the DISPLAY REGISTER command to format and display registers in the current program. The DISPLAY REGISTER command also allows you to display the values of TNS/R machine registers.

```
-PMON-DISPLAY REGISTER $PC IN H
REGISTER $PC = %H70420250
-PMON-DISPLAY REGISTER TNSR IN H

      $PC: %H70420250  $HI: %H00000000  $LO: %H00000000

 $0:        %H00000000  $AT: %H0001FFFF  $V0: %H00000002  $V1: %H0008EC03
 $4:   $A0: %H000013F4  $A1: %H700000E0  $A2: %H00000000  $A3: %H00000001
 $8:   $T0: %H0001019A  $T1: %H00000000  $T2: %H00000001  $T3: %H00000000
$12:   $T4: %H0000095E  $T5: %H0000822D  $T6: %H00000000  $T7: %H00010000
$16:   $S0: %H0000012F  $S1: %H000013E6  $S2: %H000080CD  $S3: %H00000C08
$20:   $S4: %H0008003A  $S5: %H00000006  $S6: %H00000003  $S7: %HFFFF8061
$24:   $T8: %H70000000  $T9: %H00000080  $K0: %HA713A713  $K1: %HA713A713
$28:   $GP: %H70401000  $SP: %H00001406  $FP: %H000013AA  $RA: %H704201EC
```

Note. It is recommended that you display the value of TNS/R registers in hexadecimal

ICODE

The ICODE command displays instruction mnemonics starting at the specified code address. For accelerated programs, this command can be used to list TNS instructions, TNS/R instructions, or a combination of both.

For accelerated programs, it might be easier to understand the TNS/R instruction sequence by also looking at the TNS instruction sequence that it corresponds to. This can be done using the BOTH option of the ICODE command.

For example:

```
-ICODE AT #15 FOR 3 BOTH
#15
  %001354:  > LOAD  L+002      %h7042003C:  LH    s0,4(fp)
  %001355:    LDI    +005      %h70420040:  NOP
  %001356:    IMPY             %h70420044:  SLL   s0,s0,16
  %001357:    STOR  L+001      %h70420048:  ANDI  t9,t9,0xFFDF
                               %h7042004C:  MOVE  at,s0
                               %h70420050:  ADD   s0,at,at
                               %h70420054:  ADD   s0,s0,s0
                               %h70420058:  ADD   s0,s0,at
                               %h7042005C:  SRL   s0,s0,16
                               %h70420060:  SH    s0,2(fp)
#16
  %001360:  > LDI    +001      %h70420064:  LI    s0,1
  %001361:    STOR  L+002      %h70420068:  SH    s0,4(fp)
#17
  %001362:  > LDI    +000      %h7042006C:  LI    s0,0
  %001363:    EXIT    03       %h70420070:  LI    t7,0
                               %h70420074:  LI    a1,8
                               %h70420078:  JAL   $m_EXIT
                               %h7042007C:  ADDIU sp,fp,-6
                               %h70420080:   ADDIU sp,sp,4
```

This example shows three blocks of corresponding TNS and TNS/R code. Each "block" begins with a memory-exact or a register-exact point. All intermediate code within the block is non-exact.

MODIFY REGISTER

The MODIFY REGISTER command can be used to modify the value of a specific TNS/R machine register.

SELECT DEBUGGER DEBUG

When debugging native and accelerated programs, especially at the machine level, it may be necessary or desirable to use Debug rather than Inspect. You can switch between Inspect and Debug during the same debugging session with the SELECT DEBUGGER DEBUG command. To return control of the program to Inspect, issue the Debug command "INSPECT."

For more information about the SELECT DEBUGGER DEBUG command, see Section 6, High-Level Inspect Commands.

When debugging an accelerated program at a TNS/R machine level, Debug is particularly useful for setting breakpoints on TNS/R instructions. The Inspect breakpoint list is updated from the breakpoint table when a process first enters Inspect and when control returns to Inspect from Debug; it will reflect any breakpoints you set or cleared while in Debug.

For more information, see the *Debug Manual*.

16 Using Inspect With Accelerated Programs on TNS/R Systems

Accelerated Program Debugging Overview

TNS programs, compiled on TNS/R systems by TNS compilers rather than TNS/R native compilers, also execute on TNS/R systems. TNS programs can be accelerated to run more efficiently on TNS/R systems.

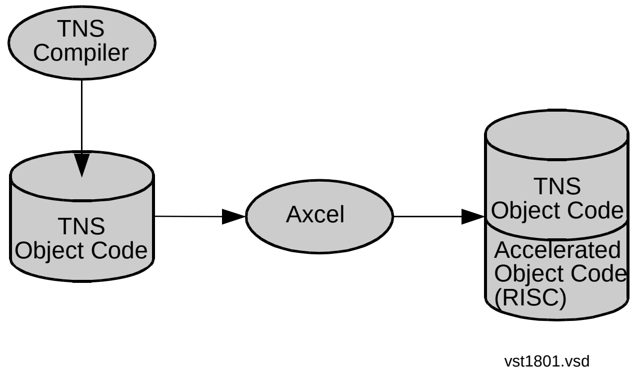
Inspect supports the debugging of both accelerated and non-accelerated TNS programs on HP NonStop TNS/R systems. Debugging non-accelerated programs on a TNS/R system is nearly identical to debugging them on a TNS system. The only difference is in the area of data breakpoints, which might be reported at slightly different locations than on a TNS system. On a TNS/R system, you can debug non-accelerated TNS programs both at the source level and at the TNS system level.

Accelerating a TNS program yields improved program performance, but can make the program more difficult to debug. When debugging an accelerated program on a TNS/R system, synchronization points—points at which there is a direct correspondence between the state of the accelerated program and the state it would have if executed on a TNS system—do not exist at all program locations. When not at a synchronization point, memory and TNS registers may be “out of date.”

Both the hardware features of the system and the optimizations performed by OCA may result in some debugging and execution differences when compared with execution on a TNS system. Differences in program execution are referred to as variances. The primary debugging difference for accelerated programs is that at some points, the program may not have the same state that it would on a TNS system.

Figure 16-1 illustrates acceleration by Axcel, the accelerator for TNS/R systems. The end product of acceleration by Axcel is a program file containing both TNS object code and accelerated (RISC) object code.

Figure 16-1. Acceleration of TNS Code on TNS/R Systems



Assumptions

If you use the Axcel accelerator to accelerate your program, you should have an understanding of the accelerator. This not only includes the operation and syntax of the accelerator, but also a conceptual understanding of its function. See the *Accelerator Manual* for additional information.

To debug accelerated programs at the machine level, you should understand the limitations and the TNS/R system variances. For more information, see Debugging Programs at the TNS/R Machine Level on page 15-10 and the *Accelerator Manual* for additional information.

Variances

Some infrequently used programming constructs do not work on TNS/R systems when a program has been accelerated. Variances are differences between TNS and TNS/R systems which can affect program execution. An example of a variance is the wrapping of user data stack addresses. For more information about variances on TNS/R systems, see the *Accelerator Manual*.

Performance and Debugging of Accelerated Programs

When you accelerate your program, instructions have been reordered and some may have been eliminated as a result of optimizations. As a result, the correspondence between TNS and TNS/R instructions has disappeared at most locations. A block of TNS/R instructions corresponding to a TNS instruction may be interleaved with other TNS/R instructions.

Listed is a summary of the primary ways that OCA improves program performance. The sections that follow explain these methods in detail and how they affect debugging.

- Retaining the values of frequently used variables in TNS/R system registers

  Inspect does not have information about when variables are stored in registers. The storing of values in registers can result in displayed memory values being out of date, memory modifications having no effect, and data breakpoints not being reported when expected.

- The elimination of unused code and the reordering of code

  Elimination of unused code can mean that you will not be able to place breakpoints at some code locations. This elimination and reordering of code also affects stepping and statement tracing, which skips any statements for which code has been eliminated or combined with the code for a previous statement.

- The elimination of branch chains

  If a program contains a branch to a label at which there is a branch to another location, the accelerator may generate code that branches directly to the target location. Such optimizations could alter the flow of control in the program.

- TNS instruction side effects

  When the accelerator optimizes code, it may eliminate TNS instruction side effects that are not used by the program. On TNS/R systems, TNS side effects such as setting the condition codes are not part of the hardware, whereas on TNS systems they are. If the logic of your program depends on the condition codes, the accelerator will generate code to preserve them; otherwise, they are not preserved.

Accelerated Program Transitions

To generate an accelerated program, the accelerator must be able to predict the possible execution paths in the program and be able to predict the value of the TNS register pointer (RP) at each location. Some TNS program constructs require evaluation at execution time and their behavior cannot be reliably predicted by the accelerator. For such constructs—such as the SETP instruction—the accelerator generates code that checks if the conditions required by the generated code are met. If they are met, execution of TNS/R instructions continues; if not, execution transfers to the corresponding TNS instructions which are executed using millicode. Execution of TNS/R instructions will resume as soon as possible.

Execution can only switch between TNS and TNS/R instructions at points where all values have been saved from the TNS/R system registers into memory and the representation of the TNS machine state is accurate. The accelerator defines such points, referred to as register-exact points, where necessary for this purpose. Register-exact points are most commonly found following procedure and subprocedure calls.

Transitions do not affect program accuracy or debugging unless you attempt to perform an operation, such as setting a TNS/R register, that depends on TNS/R instructions being executed.

Note. When accelerated programs begin execution, they begin executing TNS instructions and immediately transfer to executing TNS/R instructions.

Accelerated Program Debugging Concepts

The debugging option used with the accelerator slightly affects source-level debugging and significantly affects TNS machine-level debugging. Most source-level debugging capabilities are preserved. Source-level debugging restrictions include the possibility of deleted statements and the ability to modify memory safely. TNS machine level restrictions include the fact that breakpoints are restricted to memory-exact points and register modification is restricted to register-exact points.

Accelerated programs run faster than TNS programs, in part because the accelerated code is optimized. Traditionally, optimized code is difficult to debug because instructions can be reordered enough to blur the correspondence of instructions to source code. With the default accelerator option, ProcDebug, the accelerator optimizes over as large an area as possible to gain the maximum program performance.

The three key differences between debugging non-accelerated programs and accelerated programs are:

- Code breakpoint restrictions

  When debugging accelerated programs, code breakpoints are restricted to a subset of TNS instruction addresses. These points, referred to as memory-exact points, are the locations at which the state of displayed memory is consistent.

- Modify restrictions

  Memory-exact points, defined at the beginning of most source statements, are defined such that all writes to memory for preceding source statements have occurred. It might be the case, however, that values used by subsequent statements are already loaded in system registers. As a result, modify operations, which modify the copy in memory, might have no effect on program behavior. Register-exact points, a subset of memory-exact points, are defined so that no optimizations cross these points. Memory can be modified reliably at register-exact points. These are also points at which a program may transfer between executing TNS and TNS/R instructions.

- Data display restrictions

  The accelerator improves program performance by keeping the values of frequently used variables in TNS/R system registers. The debugger does not have information indicating when variables are stored in these registers. Such optimizations may therefore result in displayed memory values being out of date, because a more recent copy is stored in memory.

Note. If you are debugging at the TNS machine level, debug programs before accelerating for the most accurate representation of the TNS machine state.

Debugging Boundaries

Inspect provides special support to aid in debugging the optimized code generated by the accelerator. Most source-level debugging capabilities have been preserved and boundaries for debugging have been clearly defined.

Register-Exact Points

Register-exact points are locations in an accelerated program at which the values in both memory and the TNS register stack are the same as they would be if the program were executing on a TNS processor. No optimizations cross register-exact points. Complex statements might contain several such points: at each function call, privileged instruction, and embedded assignment. Register-exact points are a subset of memory-exact points.

By default, the accelerator defines register-exact points only where necessary; for example, following procedure calls. Register-exact points are the only points at which an accelerated program makes transitions to and from executing TNS instructions.

Memory-Exact Point

Memory-exact points are locations in an accelerated program at which the values in memory (but not necessarily in the register stack) are the same as they would be if the program were running on a TNS processor. Modifying memory at these points might not achieve the desired results because memory might have already been loaded in registers. Most source statement boundaries are memory-exact points. Note that register-exact points are a subset of memory-exact points.

Non-Exact Points

Non-exact points are locations in an accelerated program that are not memory-exact points. Most code locations in a accelerated program are non-exact points. At non-exact points, the TNS program state is not consistent, displayed memory may be out of date, and the reported current location is only approximate. Attempting to modify memory at a non-exact point may result in the data being overwritten by a subsequent store operation.

Accelerator Debugging Options

The accelerator debugging option you use slightly affects source-level debugging and significantly affects machine-level debugging. It affects the organization of TNS/R instructions, and the number of register-exact points in your program. The accelerator offers you two debugging options—ProcDebug and StmtDebug.

ProcDebug

ProcDebug, the default optimization level, results in optimizations that may blur statement boundaries. Blurred statement boundaries result in some statements becoming inaccessible for debugging purposes. TNS/R instructions for statements may be interleaved as long as all the stores for a previous statement are completed before the next statement. These characteristics are unique to programs accelerated with the ProcDebug option:

- Memory-exact points exist at the beginning of most statements and at some locations within statements.
- Register-exact points only occur as required by the accelerator. They often follow procedure calls within statements.
- Code optimized with the ProcDebug option produces more efficient TNS/R code than code optimized with the StmtDebug option.

StmtDebug

At this level of optimization, the TNS/R code that is generated for individual TNS instructions is not necessarily grouped in a contiguous block. Instructions may be rearranged to fill load and branch delay slots. TNS/R code for different statements is not interleaved. These characteristics are unique to programs accelerated with the StmtDebug option:

- Register-exact points occur at the beginning of most statements and at some locations within statements. As a result, the state of memory is consistent at the beginning of most statements, as is the view of the TNS machine state.
- Code optimized with the StmtDebug option is not as fast or as efficient as code generated using ProcDebug.

Summary

This table summarizes the various levels at which Accelerated programs may be debugged and your capabilities at the various levels.

Level	Required User Knowledge	Where	Capabilities
Source level	User's own program	Memory-exact points	Statement breakpoints, stepping, display variables
TNS machine level	TNS system	Register-exact points	TNS instructions breakpoints, stepping, display and modify data and TNS system registers
TNS/R machine level	TNS/R system, Accelerator	Anywhere (requires Debug to set TNS/R instruction breakpoints)	TNS/R instruction breakpoints, display and modify data and TNS/R system registers

Note. The accelerator defines memory-exact points at the beginning of most statements and at other locations where necessary. When the input object file to the accelerator contains symbols, the accelerator uses this information to determine the location of source statements. When a program does not contain symbols, locations at which the environment register RP field is 7 are treated as statement delimiters.

Table 16-1. Debugging Capabilities in Accelerated Programs on TNS/R Systems (page 1 of 2)

Action	TNS Program	Accelerated Program at a Register-Exact Point	Accelerated Program at a Memory-Exact Point	Accelerated Program at a Non-Exact Point
Add breakpoint	Yes	Yes[1]	Yes[1]	Yes[1]
Statement stepping	Yes	Yes	Yes	Yes - only to the next memory-exact point
Instruction stepping	Yes	No	No	No
Display variables	Yes	Yes	Yes	Yes[2]

Table 16-1. Debugging Capabilities in Accelerated Programs on TNS/R Systems (page 2 of 2)

Action	TNS Program	Accelerated Program at a Register-Exact Point	Accelerated Program at a Memory-Exact Point	Accelerated Program at a Non-Exact Point
Modify variables	Yes	Yes	Yes[3]	Yes[3]
Display TNS registers	Yes	Yes	Yes[4]	Yes[4]
Modify TNS registers	Yes	Yes	No	No

1 Code breakpoints can only be set on locations that are memory-exact points.
2 Data might still be in registers; therefore, displayed memory might be out of date.
3 Memory values might be preloaded in registers so modification might have not effect.
4 Register information might be out of date. i.

Using Inspect to Debug Accelerated Programs

The following subsections provide useful information for using Inspect to debug accelerated programs. Section 6, High-Level Inspect Commands provides complete syntax and detailed explanations of all commands discussed in this section.

Program Libraries

It is possible to call an accelerated user library from a program that has not been accelerated. Similarly, it is possible to call a user library that has not been accelerated from a program that has been accelerated.

When debugging such programs, the constraints and functionality that apply to debugging accelerated programs apply to the accelerated portion of the program, but not to the portion that has not been accelerated. i

Code Breakpoints

When debugging a TNS program, you can set code breakpoints at any location in the program. When debugging accelerated programs, however, Inspect only allows TNS code breakpoints to be set at locations that are memory-exact points.

If you attempt to set a TNS code breakpoint at a location that is not a memory-exact point, the following error message is issued:

```
** Inspect error 197 ** Location deleted by optimizations
```

Note. Note that “deleted” does not mean that the actions of your statements have been deleted, but that optimization has merged the statement with another which resulted in the statement not being available for debugging.

Usage Considerations for Accelerated Programs

- The INFO LOCATION command marks statement locations at which breakpoints cannot be set as “deleted.”
- When the LOCATION FORMAT command is set to STATEMENTS, statement ordinals for statements at which breakpoints cannot be set are emitted when displaying program source. For more information, see SOURCE on page 6-196.
- The SOURCE command marks deleted statements with a “-”.

Data Breakpoints

Data breakpoints, also referred to as memory access breakpoints (MABs), are available on TNS/R systems, but there are some important considerations when you use them with accelerated programs.

Data breakpoints, also referred to as memory access breakpoints (MABs), are available on TNS/R systems.

On all systems, the default type of data breakpoints is changed from WRITE to CHANGE. Change data breakpoints are like write breakpoints, except that they are only reported when the value of the variable changes.

WRITE breakpoints are not supported on TNS/R machines.

Data breakpoints may behave differently when debugging programs on a TNS/R system, especially if the program has been accelerated.

The following considerations apply when using data-breakpoints with accelerated programs.

- There is a four TNS/R instruction delay between when data breakpoints are detected and reported.

  This may result in the current program location being different from the location at which the breakpoint was detected. When this is the case, the MEMORY ACCESS BREAKPOINT event string is followed by “OCCURRED AT *src-loc*.“
- The hardware sets data breakpoints on 32-bit words. With 16-bit variables, an access to an adjacent variable may therefore trigger an unrelated breakpoint. GUARDIAN is able to filter some but not all unrelated breakpoints.

  When CHANGE breakpoints are set, unrelated breakpoints are filtered by Inspect. When READ/WRITE breakpoints are set, you must use program context information to determine which breakpoints are related to the data of interest. i

Four Cycle Delay Reporting Data Breakpoints

The pipelined TNS/R system architecture results in data breakpoints being reported four cycles after the access occurs. If a branch or procedure return occurred during that time, the current location when the breakpoint is reported could be far from the location that made the memory access. No additional data breakpoints will be triggered during the four cycle delay.

The output of the EVENT status item lists the location where the data breakpoint occurred:

```
MEMORY ACCESS BREAKPOINT bp-num:data-location [ OCCURRED AT code-location ]
```

The OCCURRED AT portion is printed if the accessing location, as represented using the LOCATION FORMAT, is different from the current location. You can use the location with the SOURCE AT command to display the source where the data access occurred.

Due to accelerator optimizations, the location reported may occasionally indicate a source line that is one or two lines before the statement accessing the data location. The actual location may be between the OCCURRED AT location and the current location.

Example

Suppose you place a breakpoint on the variable main-flag in a sample COBOL program. The memory access event will be triggered when the value of main-flag changes to one.

```
-C000OTCE-BREAK main-flag IF main-flag = 1Num Type Subtype Location  1 Data
Change  Byte Address %14252 "MAIN-FLAG" IF MAIN-FLAG = 1
```

The program is allowed to continue with the RESUME command and it encounters the data breakpoint.

```
-C000OTCE-RESUMEINSPECTMEMORY ACCESS BREAKPOINT 1: MAIN-FLAG OCCURRED AT
#LEVEL-1-PROGRAM-1.#13970(C000COBE)244,00,092  C000OTCE  #LEVEL-1-PROGRAM-
1.#13990(C000COBE)
```

Notice that the status line indicates the program has stopped at line #13990, but the memory access OCCURRED AT line #13970. The SOURCE command is used to look at this area of the source program.

```
-C000OTCE-SOURCE   #13950       Read-seq2-file SECTION.   #13960
Read-seq2-file-para.  #13970        READ seq2-file AT END  -#13980
MOVE 1 TO main-flag.  *#13990           IF (seq2-status NOT = ZEROS) AND
(NOT eof)   #14000            MOVE 2 TO main-flag.   #14010   #14020
CALL-TEST-1 SECTION.   #14030      **** Calling level-2-program-1 with 2
BY REFERENCE and 2 BY CONT   #14040           PERFORM test-init.
```

The current location, #13990, is indicated by the “*”. This is where the program is currently suspended.

The actual memory access occurred earlier at line #13970, according to the OCCURRED AT clause. Looking at the source, it does not appear that the value of main-flag would have changed in line #13970. The actual place where the memory access occurred is somewhere between the reported (approximate) location, line #13970, and the next exact point (note that line #13980 is deleted, so the next exact point is line #13990).

Therefore, in this case the actual memory access occurred somewhere during execution of lines #13970 or #13980. Looking at the source, it becomes obvious that the actual memory access of main-flag occurred on line #13980.

The following illustrate a number of anomalies that can occur when the variable that triggers a data breakpoint is accessed again during the four cycle delay.

Note. The anomalies discussed below are unlikely to occur unless simple assignment statements are used in succession.

This pseudocode illustrates several of these anomalies:

```
#110    X :=1
#120    X :=2
#130    X :=3
```

- Data breakpoints

  A breakpoint is set on the data item X prior to reaching line #110. When the program reaches #110, the data breakpoint cycle will be triggered. When the program stops four cycles later, the current location might be line #130 with a report that the data breakpoint took place at line #110. Examining X might show 2 or 3 rather than the 1 stored at line #110.

- Code breakpoints and data breakpoints

  If a code breakpoint is at the same location that accesses memory on which a data breakpoint is set, the code breakpoint is reported first.

  Using the same pseudocode as before and the same data breakpoint on X, consider the results of putting a breakpoint at line #120. When the program reaches line #110 the data breakpoint cycle will start; however, it will not complete before the code breakpoint at line #120 is encountered. This will give the appearance that the data breakpoint did not occur even though an examination of the data shows the value has been stored. Attempting to STEP or RESUME from this point will release the data breakpoint cycle and the data breakpoint will eventually be reported.

- Stepping

  Using the same pseudocode shown above, a data breakpoint is set on X prior to line #110. Stepping a line at a time starting at #110 might not report that breakpoint occurred at #110 until you reach line #120 or #130.

- Data breakpoints of type change

  There may be occasions when data breakpoints will not be reported. Consider the following pseudocode:

```
#100   X :=1
#110   Y :=1
#120   X :=3
#130   Y :=X
#140   X :=1
```

  If the current location of Inspect within the program is line #110 and a change data breakpoint is set on X (note that change is the default for data breakpoints), when the program is resumed the data breakpoint cycle will begin when the store to X occurs at line #120. Before the four cycles have elapsed, X might be restored to a value of 1 and the data breakpoint will not be reported because it appears that the data did not change.

Accesses to 32-Bit Words

The TNS/R system reports data breakpoints on accesses to 32-bit words. For variables that occupy fewer than 32 bits, this can result in unrelated breakpoints being reported when an access is made to the other portion of the 32-bit word.

When data breakpoints of type change are used, Inspect can filter out such unrelated breakpoints, reporting only when the value of the variable has changed. When read/write data breakpoints are used, data breakpoints may be reported as a result of memory accesses to the 32-bit word containing the variable. i

Data Breakpoints

The following considerations apply to data breakpoints:

- The default type for high-level data breakpoints is change.

  This applies to both TNS and TNS/R processors. Data breakpoints of type change are only reported if the value of the variable has changed; writes that store the same value already contained in the variable are not reported. Use of the change breakpoint allows Inspect to filter all unrelated breakpoints.

- Write breakpoints are available on TNS processors, but not on NSR-L processors.

  You receive the following error message if you attempt to set a write data breakpoint in a program executing on a NSR-L processor:

```
** Inspect error 365 ** Machine does not support write memory access
breakpoints
```

  Because write breakpoints are not supported on NSR-L processors, use change breakpoints instead.

- Read/write data breakpoints are available on both TNS and TNS/R processors.

  When data breakpoints are set on read/write access on a TNS/R system, the operating system or Inspect cannot filter unrelated accesses. You will have to use context information associated with the program's current location to determine breakpoints of interest.

Inspect issues the following warning message if you set a data breakpoint on a variable that is larger than the 16-bit word size of the TNS system:

```
** Inspect warning 363 ** Breakpoint is set on the first 16-bit word
                            containing the variable
```

Inspect issues the following warning message when you set a write data breakpoint on a variable that is smaller than the word size of the system (which is 32 bits on a TNS/R system):

```
** Inspect warning 127 ** Breakpoint will occur upon access to containing
word
```

Usage Considerations for Accelerated Programs

The following additional considerations should be noted when using data breakpoints with accelerated programs:

- If an instruction accesses a memory location on which a data breakpoint is set, any memory accesses made by subsequent instructions before the original breakpoint is reported are not detected.
- Data breakpoints most likely suspend the program at non-exact points; that is, locations that are not memory-exact points.
- Some read data breakpoints may not be reported because no memory operation has taken place. The accelerated program may keep values of frequently used variables in registers, avoiding the need to go to memory to fetch them, which then results in those read data breakpoints not being reported.

COBOL85 Example

When a COBOL object file is accelerated, the accelerator often has a wide range of instructions over which to apply optimizations since many COBOL constructs, such as INITIALIZE, SEARCH, or UNSTRING, require a large number of instructions to implement. This optimization may affect debugging in the following ways:

- It may reduce the correspondence between the generated instructions and source constructs.
- It also increases the likelihood that a data breakpoint will leave the program at a non-exact point.

Some COBOL constructs, such as IF or AT END, may not result in memory-exact points, and section and paragraph names may not result in register-exact points even if the accelerator optimization level is set to StmtDebug. Therefore, it may not be possible to set code breakpoints at some such locations or to use the RESUME AT command.

This example shows the effect ProcDebug optimizations can have on your program and reaching a non-exact point.

```
-C000TTPT-BREAK main-flag if main-flag = 1
Num Type Subtype Location  1 Data Change  Byte Address %11750 "MAIN-FLAG"
IF MAIN-FLAG = 1
-C000TTPT-RESUME
INSPECT
MEMORY ACCESS BREAKPOINT 1: MAIN-FLAG OCCURRED AT #LEVEL-1-PROGRAM-
1.#13970(C000COLD) 244,01,083 C000TTPT  #LEVEL-1-PROGRAM-
1.#13990(C000COLD) ** Inspect warning 359 **** Current location is not a
memory-exact point;
                     displayed values may be out of date;
                      the location reported is an approximate TNS
location-C000TTPT-SOURCE
#13950       Read-seq2-file SECTION.
#13960       Read-seq2-file-para.
#13970           READ seq2-file AT END
 -#13980             MOVE 1 TO main-flag.
 *#13990           IF (seq2-status NOT = ZEROS) AND (NOT eof)
#14001             MOVE 2 TO main-flag.
#14010
#14020       CALL-TEST-1 SECTION.
#14030   *** Calling level-2-program-1 with 2 BY REFERENCE and 2 BY CONTE
#14040           PERFORM test-init.
```

This shows the result of accelerating the code with StmtDebug:

```
-C000TTPT-BREAK main-flag if main-flag = 1
Num Type Subtype Location
   Data Change  Byte Address %11750 "MAIN-FLAG" IF MAIN-FLAG = 1
-C0000TCE-RESUME
INSPECT
MEMORY ACCESS BREAKPOINT 1: MAIN-FLAG OCCURRED AT #LEVEL-1-PROGRAM-
1.#13980(C000COBE)244,00,092  C000OTCE  #LEVEL-1-PROGRAM-
1.#13990(C000COBE)
-C000TTPE-SOURCE
 #13950       Read-seq2-file SECTION.
 #13960       Read-seq2-file-para.
 #13970           READ seq2-file AT END
@#13980             MOVE 1 TO main-flag.
*#13990           IF (seq2-status NOT = ZEROS) AND (NOT eof)
@#14000             MOVE 2 TO main-flag.
 #14010
 #14020       CALL-TEST-1 SECTION.
 #14030      **** Calling level-2-program-1 with 2 BY REFERENCE and 2 BY
CONT
 #14040           PERFORM test-init.
```

Event Reporting

The consistency of the state of an accelerated program depends on the current location when a debugging event is reported. Code breakpoints and many other debugging events are reported when the current location is either a memory-exact point or a register-exact point.

There are some debugging events that may be reported when the current location is a non-exact (not a memory-exact) point: i

- A data breakpoint.
- A process entry into Inspect.
- A run-time trap.
- An INSPECT HOLD command issued to suspend a running process.
- A TNS/R code breakpoint. (TNS/R code breakpoints may only be set in Debug.)

When a debugging event occurs, Inspect determines the consistency of the program state at the current program location. The current location can be one of the following:

- Register-exact point

  Inspect presents you with the most information about the TNS view of the current program state at register-exact points. At a register-exact point, TNS registers and memory are consistent.

- Memory-exact point

  Inspect presents you with less up to date information at memory-exact points. At a memory-exact point, displayed memory is accurate and some TNS register values may be out of date.i

- Non-exact point

  Inspect presents you with the least accurate information at non-exact points. The reported current program location is only approximate. Displayed memory may be out of date, memory modifications may have no effect, and TNS registers are out of date.

  If your current program location is a non-exact point, Inspect will issue the following warning when a debugging event is reported:

```
** Inspect warning 359 ** Current location is not a memory-exact
point;displayed values may be out of date; the location reported is an
approximate TNS location
```

Data Access Limitations

The following paragraphs describe data access limitations that exist at various points in your program.

At Register-Exact Points

When the current program location is a register-exact point, all memory has been updated and no data is loaded into TNS/R registers. You can display and modify variables without concern that memory is out of date.

At Memory-Exact Points

When the current program location is a memory-exact point that is not a register-exact point, you can display memory with reliable results.

The accelerator defines memory-exact points such that all preceding memory store operations have completed. Unless the point is a register-exact point, values used by subsequent statements might already be loaded in TNS/R registers. In this case, modifying the value of the variable in memory has no effect. Inspect has no way of determining when this is the case; therefore, the commands that modify memory, MODIFY and low-level M, report the following warning when the current location is a memory-exact point:

```
** Inspect warning 358 ** Modify may have no effect; data that is about to
be used may be stored in registers
```

As a guideline, values that have been accessed recently, are just about to be accessed, or are accessed frequently are likely to be stored in registers.

At Non-Exact Points

When the current program location is a non-exact point, the TNS program state cannot be mapped directly from TNS/R to TNS machine locations. Displayed memory may be out of date. Attempting to modify memory at a non-exact point has the same limitations as at memory-exact points.

In addition, it may result in the data being overwritten on a pending store operation.

TNS Register Access Limitations

The following paragraphs describe TNS register access limitations that exist at various points in your program.

At Register-Exact Points

Register-exact points are locations in an accelerated program at which the values in both memory and the register stack are up to date. You can both display and modify TNS registers at register-exact points.

Even at register-exact points, the value of a given register will match what you would see at the same point if running on a TNS system only if the register value is used by subsequent TNS instructions. For example, TNS load instructions cause the condition code to be set (N and Z bits in environment register), but the accelerated code will only set the condition code if the program actually references it.

Note. The usefulness of modifying TNS system register values is extremely limited because of the limited number of register-exact points.

At Memory-Exact Points

You can display values of TNS registers at a memory-exact point; however, displayed values may be out of date.

- Commands that display register values (DISPLAY and low-level D) report the following warning:

  ```
  ** Inspect warning 354 ** Register values are out of date
  ```

- Commands used to modify registers (MODIFY REGISTER, and low-level M) report the following error when at a memory-exact point that is not a register-exact point:

  ** Inspect error 355 ** Register values are out of date; they cannot be modified

At Non-Exact Points

The memory-exact point restrictions apply when the current location is not a memory-exact point.

Commands Useful When Debugging Accelerated Programs

The Commands explained are useful when debugging Accelerated programs. For more information, see Section 6, High-Level Inspect Commands.

INFO LOCATION

The INFO LOCATION command gives information about a statement in the current program. If you are executing an accelerated program, displayed information includes the effects that accelerator optimizations have on source statements.

Example

You can use the INFO LOCATION command to determine your program’s current location. A “Yes” in the Register-Exact column indicates that the beginning of the source line is a register-exact point.

```
-SERVER-INFO LOCATION #78Scope: MCompile File:  $TOOLS.SRVSRC.SVRMN
Modified:  1988-10-13 12:15:12.98                    Word
Register-                          Num   Line          Offset   Optimize
Verb  Exact                         12    #78         %33
Yes
```

Note. When the current executing TNS/R code location is a non-exact point, the reported source line location is approximate. To advance to the next memory-exact point, use the STEP command.

INFO OBJECTFILE

The INFO OBJECTFILE command gives information about the current program's object file. If you are running an accelerated program, displayed information includes the options with which your program was accelerated and when it was accelerated.

Example

This example shows, the INFO OBJECTFILE command was used to show that the program was accelerated with the ProcDebug option, which provides more optimizations but less debugging information than the StmtDebug option.

```
-PROGRAM-INFO OBJECTFILEObject File: \SYS.$VOL.SUBVOL.PROGRAM
General Information                 BINDER Region: YES          BINDER
Timestamp: 1992-08-13 17:46:40.57               Data Pages: 64
Debugger: INSPECT                INSPECT Region: YES                System
Type:  GUARDIAN               Process Subtype: integer        Program File
Segment: 0 WORDS                Highrequesters: OFF
Runnamed: OFF                       Highpin: OFF
Saveabend: OFF                      Segments: 1                      Target:
UNSPECIFIED                   Accelerator Information       Accelerated
Execution: ENABLED                  Optimization: PROCDEBUG
Global Options: ATOMIC_OFF, INHERITSCC_OFF, OVTRAP_ON,
SAFEALIASINGRULES_OFF, TRUNCATEINDEXING_ON                 Timestamp:
1992-08-13 18:29:17.52                    Version: 1992-02-25 10:18:32.46
```

LIST PROGRAM

The LIST PROGRAM command identifies the type of program—TNS/R or TNS:

```
                 ProgramNum  Program ID  Name       Type         State
Location *1      06,081  OTALIN     TNS          HOLD  #M.#8   2
07,172  PTALIN     TNS/R        HOLD  #M.#64
```

When debugging an accelerated program, some of the information listed by the LIST PROGRAM DETAIL command may be of particular interest, including an indication that the program has been accelerated and:

- The instruction set

  The instruction set indicates whether TNS or TNS/R code is currently executing. TNS is listed for an accelerated program when the program temporarily transfers to executing TNS instructions. "Type" indicates what type of program it is and "instruction set" indicates what code is currently executing.

- The accelerator state

  The accelerator state indicates whether the location where the program is currently suspended is a register-exact, memory-exact, or non-exact point. This information indicates information you need to determine what debugging actions will provide reliable results. For example, if you are at a memory-exact point, you can display memory reliably, but not modify. The accelerator state is not listed if an accelerated program is executing TNS instructions.

When the accelerator state is at a memory-exact or a register-exact point, the reported current location is exact. When the accelerator state is a non-exact point, the current location is a approximate. This means that the program is suspended between the reported location and the next exact point.\

Example

The accelerator state shows that the program is at a memory-exact location and the instruction set TNS/R is displayed to indicate that the accelerated program is executing TNS/R instructions.

```
-C000TTST-LIST PROGRAM *, DETAILName: C000TTSTNumber: 1
General Information           Accelerated: YES      Accelerated State:
Memory-exact              CPU,PIN: 0,89       Guardian Version: C30
Instruction Set: TNS/R              Location: #LEVEL-1-PROGRAM-
1.#13990(C000COLD)            Processor: TNS/R (NSR-L)          Program
File: $OCT.QAT9257I.C000TTST            Library File:
$OCT.QAT9257I.C000OBF3                   State: HOLD
System: \BASTILL (244)                    Type: TNS/R            INSPECT
Information       ABEND Breakpoint: NO       Code Breakpoints: 0
Data Breakpoints: 1          Source System: None        STOP Breakpoint:
NO
```

RESUME AT

When debugging accelerated programs, you cannot transfer program execution to arbitrary program locations (as you can on TNS machines). The source and destination locations must both be register-exact points.

As with TNS systems, you need to ensure that the value of the TNS register pointer (RP) is the same at both locations. RP usually has a value of seven at the beginning of most statements.

Note. When a program has been accelerated with the ProcDebug option, the usefulness of the RESUME AT command is constrained by the limited number of register-exact points at the beginning of statements. The StmtDebug option, however, results in register-exact points at the beginning of most statements.

If the target location is not a register-exact point, the following error message is reported:

```
** Inspect error 357 ** Target location must be a register-exact point
```

If your current location is not a register-exact point, the following error message is reported:

```
** Inspect error 370 ** Current location must be a register-exact point
```

Note. The register-pointer (RP) clause of the RESUME command is not supported for accelerated programs.

SET PROMPT/SET STATUS

The following status tokens could be helpful when used with the SET STATUS and SET PROMPT commands:

- ACCELERATOR STATE
- INSTRUCTION SET
- PROCESSOR

Examples

- This example shows, how you might set your prompt to include accelerator state information:

```
SET PROMPT = LEVEL,PROGRAM NAME,"[ ",ACCELERATOR STATE, " ]", STEP, LEVEL
```

This will result in prompts of the following form:

```
-ATMSRV[ memory-exact ]-
```

In this case, the location in TNS/R code where the program suspended is a memory-exact point.

SOURCE

When debugging accelerated programs on TNS/R systems, the SOURCE command annotates statements that are register-exact points and those that have been deleted (that is, statements that are not memory-exact points). The annotation character is listed in the column before the line/statement number: i

Character	Description
–	The statement is “deleted” (it is not a memory-exact point).
@	The statement is a register-exact point; the RESUME AT command and register modification commands can be used at such statements.

Usage Consideration for Accelerated Programs

The annotation character is listed in the same column that Inspect lists the asterisk when marking the current location; the asterisk always takes precedence. Information about the program state at the current location is available from the ACCELERATOR STATE status and prompt tokens and the LIST PROGRAM DETAIL command.

Example

The following example illustrates use of the SOURCE command to determine your program's current location and identify points that are register-exact or memory-exact points.

```
-PTALIN-SOURCE FOR 10*#75.1      i := 1;
#76          a := 1;
#77          CALL s1;
@#78          CALL s2( a, a, a );
@#79          CALL s3;
-#80          b := a;
#81
#82          CALL PROC1( 1, 2D, sptr );
@#84           CALL PROC2( 1, 2D, sptr ); #84.1
```

STEP

The STEP command allows you to STEP statements and verbs in accelerated programs, but not instructions. If you attempt to STEP instructions in an accelerated program, Inspect issues the following error message:

```
** Inspect error 353 ** Current location is accelerated; stepping
instructions is not allowed.
```

Usage Considerations for Accelerated Programs

Stepping the execution of an accelerated program is subject to the following considerations:

- The STEP command always leaves an accelerated program at a memory-exact point. (Note that register-exact points are a subset of memory-exact points.)
- When the current program location is not a memory-exact point, entering a STEP or STEP IN command advances the program to the next memory-exact point. In some instances, this could be in the middle of a source statement. If you have been advanced to the next memory-exact point Inspect will issue the following warning message:

```
** Inspect warning 391 ** Current location is not a memory-exact
point; execution will be stepped to the next memory-exact point
```

- A multi-statement or multi-verb step command does not always produce the same results as issuing an equivalent number of single statement/verb commands.

  If one or more statements at the end of the step range have been deleted, stepping continues to the next memory-exact point. However, a multi-statement or multi-verb step is not affected by deleted statements that are contained within the step range.
- Transitions between TNS and accelerated code execution do not affect stepping.

- Accelerating a program may cause fewer step operations to be required to step execution through some looping constructs. Depending on compiler code generation, the accelerator may branch execution to the middle of a statement, where there is no memory-exact point. Since STEP operations always leave a program at a memory-exact point, execution of the program will not be suspended until the next memory-exact point is reached. Usually, this is the next statement.

Example

The following example illustrates that Inspect STEP behavior within loops may differ between TNS programs and accelerated programs, as illustrated with using this sample code written in C.

```
Statement
Number           func1()
* 1         {
            int i,j,k;
  2         k = 1;
  3         for (i=0; i<2; i++) {
  4           j =  50 * i;
  5           k += 100 + j;
  6           }  /* end for */
  7         printf ("i=%d, j=%d, k=%d\n", i, j, k);
  8         }
```

The following illustrates the result of issuing successive STEP commands. Each line shows the line from the preceding source that execution has been advanced to.

```
Stepping a TNS program         Stepping an Accelerated program
1   {                             1   {
2     k = 1;                      2     k = 1;
3     for (i=0; i<2; i++) {       3     for (i=0; i<2; i++) {
4       j =  50 * i;              4       j =  50 * i;
5       k += 100 + j;             5       k += 100 + j;
6       }  /* end for */          6       }  /* end for */
3     for (i=0; i<2; i++) {
4       j =  50 * i;              4       j =  50 * i;
5       k += 100 + j;             5       k += 100 + j;
6       }  /* end for */          6       }  /* end for */
3    for (i=0; i<2; i++) {
7     printf ("i=%d, j=%d,        7     printf ("i=%d, j=%d,
        k=%d\n", i, j, k);                 k=%d\n", i, j, k);
```

Note, in the accelerated program, Inspect does not stop on statement 3 after the first time through the loop. At the locations where it stops in both the TNS and TNS/R code, both programs will be in the same state. For example, displaying the value of I while at the second instance of statement 4, yields the same results for both programs.

Annotated ICODE

When debugging accelerated programs, TNS instruction code mnemonics listed by the SOURCE ICODE and ICODE command and the low-level I command are annotated to indicate which locations are register-exact points (@) and which are memory-exact points (>).

When debugging accelerated programs, TNS instruction code mnemonics listed by the SOURCE ICODE and ICODE command and the low-level I command are annotated to indicate which locations are register-exact points (@) and which are memory-exact points (>). i

Character	Description
@	The instruction is at a register-exact point.
>	The instruction is at a memory-exact point.

Examples

- The SOURCE ICODE command could produce the following output:

```
#417            CALL ER^Write( ERR^NOT^DISC^FILE, 0, , fname^ext,
fname^len      LDI    +101        ZERD              LADR  L+014,I
LOAD  L+015        ZERD              LDI    +154      PUSH    777
XCAL    070#418          fn := -1;       LDI    -001         STOR
L+013#419          CALL OPEN( fname^int, fn, OPEN^READONLY );
LADR  L-003,I      LADR  L+013       LDLI   +004      PUSH    722
ADDS   +006        LDLI   +340     LDI    -011        PUSH    711
XCAL    257
```

- For example, the low-level I command could produce the following output:

```
000006:   > LOAD    L+006    STOR    L+005     LOAD    L+006000011:
CMPI      004    BNEQ      010   @ LDI      +007000014:     STOR    L+006
> LDI       010     STOR    L+002
```

17 Using Inspect With TNS/R Native Programs

TNS/R Native Overview

Three modes of execution are possible on a TNS/R system: TNS/R native mode, TNS mode, and accelerated mode.

Much of the code in HP-supplied software products for TNS/R systems has been produced by TNS/R native compilers. Users can also use TNS/R native compilers to produce their own native TNS/R code. (Refer to the *C/C++ Programmer's Guide* and the *pTAL Reference Manual*.) TNS/R native code consists of RISC instructions that have been optimized to capitalize on the RISC architecture. Program files containing such code are called native program files.

TNS programs, produced by TNS compilers executing on TNS/R systems, also can execute on TNS/R systems. TNS programs contain TNS object code, and program files containing TNS object code are called TNS program files.

TNS/R Native Program Debugging Concepts

TNS/R native programs run faster than accelerated or TNS programs, in part because the native code is optimized. The optimization level option used when compiling TNS/R native programs directly affects source-level debugging. (Most source-level debugging capabilities are preserved. Source-level debugging restrictions include deleted statements and the inability to modify memory safely.) Traditionally, optimized code is difficult to debug because instructions can be reordered enough to blur the correspondence of instructions to source code.

Debugging a TNS/R native program is similar to debugging the RISC portions of an accelerated program, but you should be aware of a few differences.

- In TNS/R native mode, local variables are sometimes cached in registers. Attempting to modify a local variable or use it for setting a memory access breakpoint, for example, can have unexpected results.
- In highly optimized TNS/R native object code, parameter values are sometimes cached in registers, making their exact location unpredictable.
- In TNS/R native mode, unlike accelerated mode, there are no TNS instructions corresponding to the RISC instructions.
- Inspect allows you to display register-based variables in any stack frame. However, Inspect only allows modification of register-based variables if the current scope is active, that is, the top of the stack.
- Inspect creates and reads save files from TNS/R native processes. This includes saving stack frames, Global data and heap areas, main stack, and SRL instance data. Inspect also includes the trap or signal number in the save file if the application ABENDed due to a trap.
- To display TNS/R frames, use the TRACE REGISTERS command. For TNS/R frames, Inspect displays the PC register in the normal Inspect form as a scope name and offset. Inspect also displays the virtual frame pointer (VFP) for that frame.
- To set a breakpoint in TNS/R native code, you can either specify a scope name or a machine-level code address.

TNS/R Native Compilers and Linkers

TNS/R systems support the following native compilers (manuals are listed also):

C/C++	*C/C++ Programmer’s Guide*
COBOL85	*COBOL85 for NonStop Systems Manual*
pTAL	*pTAL Reference Manual*

These compilers produce only RISC instructions.

On TNS/R systems, Binder is replaced by the native link editor (nld) and the native object file tool (noft). Additionally, for position-independent code (PIC) programs on TNS/R, you must use the ld linker in place of nld, and also use the PIC run-time loader, rld. For more information, see the *nld and noft Manual* and the *ld and rld Reference Manual*.

To debug a TNS/R program that contains position-independent code (PIC), you must use Visual Inspect.

Optimization Levels

Debugging TNS/R native processes differs from debugging TNS or accelerated processes. Unlike TNS and accelerated processes, TNS/R native processes do not maintain the TNS process environment, such as the TNS environment registers.

The C, C++, and pTAL compilers support three optimization levels: 0, 1, and 2. Source-level debugging capabilities are directly related to the optimization level of the compiler.

Optimize 0

At this level of optimization, no optimizations occur. There is no scheduling, all delay slots are filled with a NOP, and memory is immediately updated after calculations. That is, variables are placed in registers, necessary calculations are performed, and the variables are immediately updated.

Optimize 1

At this level of optimization, as many delay slots as possible are filled and updating of memory is sometimes deferred because of forward motion of store instructions by the scheduler.

Optimize 1 offers full debugging support. However, statement boundaries can be blurred. Inspect will choose a sensible location when you request a breakpoint on a source statement. Inspects statement boundaries do not necessarily coincide directly with source statements and will emit a warning when a process is held (e.g., hits a breakpoint) at a statement for which some code associated with a previous source statement, e.g. a store instruction, has yet to execute.

Optimize 2

At this level of optimization, the compiler performs all of its possible optimizations– register allocation, code motion, and instruction scheduling. The register allocation algorithm cannot be tracked with the current debugger symbol table information. Practically, there is no symbolic debugging support for Optimize 2.

Table 17-1. Optimization Levels

Level	Optimization Performed	Debugging Capabilities
0	None. Variables are loaded into registers, calculations are performed, and variables are immediately updated in memory. Statement boundaries are maintained. Delay slots in instructions sequence are filled with no-operation instructions.	Full source-level debugging capability. Inspect sets code and data breakpoints and steps similar to TNS processes.
1	Most optimizations performed. Statement boundaries are blurred. Instructions are moved forward and backward to fill delay slots.	Nearly full source-level debugging capability. Inspect makes a best-guess at where statements begin and end. Inspect emits a warning if finds a statement for which some code associated with a previous statement that has yet to execute.
2	The compiler performs all of its possible optimizations.	Limited source-level debugging capabilities. Statements may have been deleted and/or merged with other statements.

The compiler performs most optimizations. Statement boundaries are blurred as the compiler moves instructions corresponding to one statement into forward or backward statements. Inspect emits a warning if it finds a statement containing code for a backward statement that has not been executed.

Using Inspect to Debug TNS/R Native Programs

The following subsections provide information about using Inspect to debug TNS/R native programs. Section 6, High-Level Inspect Commands, provides complete syntax and detailed explanations of all commands discussed in this section.

SRLs

Inspect supports native SRLs (shared run-time libraries) on TNS/R systems. You can set code and data breakpoints in SRLs per process, create save files for applications that use SRLs, and display and modify identifiers from SRLs.

An SRL is an object file that the operating system links to a program file at run time. C, C++, TAL, and pTAL programs can have a user library. On a TNS/R system, there are two types of user libraries: TNS user libraries and TNS/R native user libraries. TNS user libraries can be called by TNS and accelerated processes. TNS/R native user libraries can be called by TNS/R native processes.

TNS/R native user libraries are implemented as a special form of a native shared run-time libraries (SRLs). From a debugging perspective, this special form of SRL behaves the same as a TNS user library.

The process memory architecture and implementation of TNS/R native user libraries differs from TNS user libraries. The distinction between user code and user library space (UC and UL) does not exist in TNS/R native processes. There is one address space for TNS/R native processes. Because of this difference in process memory, you no longer need to specify a code location as user code (UC) or user library (UL).

By default, Inspect will not load public SRLs. Inspect does load ULs and private SRLs. Use the SELECT PROGRAM command to load SRL files.

For more information on TNS user libraries, see the *Binder Manual*.

For more information on TNS/R native user libraries, see the *nld Manual and noft Manual*.

Dynamic-Link Libraries (DLLs)

Inspect does not support debugging of any TNS/R native process that uses Dynamic-Link Libraries (DLLs). To debug a process that uses DLLs, you must use either Visual Inspect or Debug.

For more information about programming with DLLs, see the *DLL Programmer's Guide for TNS/R Systems*.

Example

Inspect does not automatically load the symbol files for Public SRLs. The ADD PROGRAM and SELECT PROGRAM command can be used to load an SRL if necessary. For example, to set a break point in the function printf. Using the SELECT PROGRAM command, you can load the SRL which contains the printf function, and set the break point.

```
-PROG-MATCH SCOPE printf
No scopes found that match printf
-PROG-SELECT PROGRAM 1 SRL ($SYSTEM.SYS00.ZCRTLSRL)
               Program
Num Program ID Name      Type        State Location
 *1   00,00034 ETEST      TNS/R        HOLD  #main.#
-PROG-MATCH SCOPE printf
Library Procedures (\CHIMP.$SYSTEM.SYS00.ZCRTLSRL):
printf
-PROG-BREAK #printf
Num Type Subtype Location
  2 Code        #printf.#73
-PROG-RESUME
INSPECT  TNS/R BREAKPOINT 2: #printf
241,00,00034  ETEST  #printf.#73
-PROG-TRACE
Num Lang Location
  0  C   #printf.#73
  1  C   #main.#13
  2  C   #_MAIN.#36
-PROG-
```

Code Breakpoints

When debugging a TNS/R program, you can set code breakpoints at any location in the program. To set a breakpoint in native code, you can either specify a scope name or a machine-level code address.

Signals

TNS and accelerated processes use a trap mechanism for exception handling. TNS and accelerated processes in the OSS environment can also use a signal mechanism for exception handling. (A signal is a means by which a process can be notified of or affected by an event occurring in the system.)

TNS/R native processes, regardless of whether they are in the Guardian or OSS environment, use a signal mechanism for exception handling. (OSS processes can also use signals for communicating between processes.) The SIG_DEBUG parameter was added to the MODIFY SIGNAL command to specify that when a signal is delivered, the debugger will be invoked.

For example, where a TNS process might enter a trap handler on an arithmetic overflow, a TNS/R native process might enter a signal handler.

Refer to the *Guardian Programmer’s Guide* and the *Open System Services Programmer’s Guide* for details on signal handlers.

Commands Useful When Debugging Native Programs

The commands listed are beneficial when debugging TNS/R native programs. For more information, see Section 6, High-Level Inspect Commands.

ADD PROGRAM

The ADD PROGRAM command accepts the names of one or more SRLs. The SRL clause is a single filename, or a comma separated list of SRL object files within parentheses. If the process used more SRLs than you specified, Inspect will emit a warning indicating this. By default, Inspect does not load public SRL object files.

BREAK

The BREAK command supports memory access breakpoints on TNS/R native addresses. The support of MABs on native addresses applies to all data breakpoint subtypes (ACCESS, CHANGE, READ, WRITE, READ WRITE, WRITE READ).

DISPLAY REGISTER

The DISPLAY REGISTER command displays the virtual frame pointer (VFP) for TNS/R native stack frames. In addition, the contents of the registers will be relative to the current stack frame. This means that as the current scope is changed to an active stack frame, the register values will be what they were in that stack frame. If the current scope is changed to a non-active stack frame, the register values will be what are current in the program. Note that not all registers are saved in every procedure. This may result in some registers having values left over from the preceding stack frame.

Output

This example output shows all registers.

```
-PROGRAM-DISPLAY REGISTER ALL

 $PC:  1879048880  $HI:     926     $LO: 3976303904  VFP:  1342177056

 $0               0 $AT: 134217732  $VO:          0  $V1:          1
 $4: $A0          0 $A1          0  $A2   134230564  $A3:         92
 $8: $T0  134230560 $T1          4  $T2          66  $T3   1879082720
$12: $T4      65043 $T5     262144  $T6           0  $T7           0
$16: $S0  4294967295 $S1 4294967295  $S2  4294967295  $S3   4294967295
$20: $S4  4294967295 $S5 4294967295  $S6 4294967295  $S7   4294967295
$24: $T8  134230656 $T9           2 $K0  2803083027  $K1   2803083027
$28: $GP   134254384 $SP  1342177016  $FP 4294967295  $RA   1879082720
```

ICODE

The ICODE command displays TNS/R native instructions.

RISC instructions from a previous source line are annotated with a “-” and RISC instructions from proceeding lines are annotated with a “+”. Lines containing RISC instructions also contain the source file line number that the instruction is for.

```
-PROGRAM-ICODE AT #PROC FOR 3 STATEMENTS

 #10
   10.000    %h700002B0:  addiu   $sp,$sp, -32
   10.000    %h700002B4:  sw      $4,32($sp)
   10.000    %h700002B8:  sw      $5,36($sp)
 #17
   17.000    %h700002BC:  lw      $15,36($sp)
   17.000    %h700002C0:  lw      $14,32($sp)
 - 10.000    %h700002B4:  sw      $6,40($sp)
   17.000    %h700002C8:  lw      $25,40($sp)
 - 10.000    %h700002CC:  sw      $7,44($sp)
   17.000    %h700002D0:  lw      $9,44($sp)
   17.000    %h700002D4:  add     $24,$14,$15
 + 18.000    %h700002D8:  sw      $11,$48($sp)
   17.000    %h700002DC:  add     $8,$24,$25
 - 10.000    %h700002E0:  sw      $31,28$sp)
   17.000    %h700002E4:  add     $10,$8,$9
```

```
    17.000    %h700002E8:  lw      $10,16($gp)
 #18
    18.000    %h700002EC:  move    $5,$15
    18.000    %h700002F0:  move    $4,$14
    18.000    %h700002F4:  move    $6,$15
    18.000    %h700002F8:  move    $7,$9
    18.000    %h700002FC:  jal     0x70000290
    18.000    %h70000300:  sw      $11,$16($sp)
-PROGRAM-
```

INFO IDENTIFIER

The INFO IDENTIFIER command displays information about TNS/R native symbols. Relevant information includes: parameters and local variables will be relative to the virtual frame pointer (VFP), not the L register, and register based variables will display the RISC register name.

INFO OBJECTFILE

The INFO OBJECTFILE command displays information from TNS/R native object files.

Output

This output template shows the information of the INFO OBJECTFILE command displays.

```
[ Library | Program]Object File: filename

                General Information

[           BINDER Region:  YES | NO                          ]  *1
[        BINDER Timestamp:  timestamp                         ]  *1
[           NLD Timestamp:  timestamp                         ]  *3
[              Data Pages:  integer                           ]  *1
                 Debugger:  DEBUG | INSPECT
           INSPECT Region:  YES | NO
              System Type:  GUARDIAN | OSS
          Process Subtype:  integer
     Program File Segment:  integer WORDS
           Highrequesters:  ON | OFF
                 Runnamed:  ON | OFF
                  Highpin:  ON | OFF
                Saveabend:  ON | OFF
                 SRL Name:  name                               ]  *4
                 Segments:  integer                            ]  *1
                   Target:  target                             ]  *1

[             Accelerator Information                          ]  *2
[]*2
[   Accelerated Execution:  ENABLED | DISABLED                 ]  *2
[            Optimization:  PROCDEBUG | STMTDEBUG | UNKNOWN    ]  *2
[          Global Options:  accelerator options                ]  *2
[               Timestamp:  timestamp                          ]  *2
[                 Version:  timestamp                              ]  *2
```

If the filename parameter is omitted, Inspect will display information about all object files used by the current process (user code plus all libraries plus all system files). Items labeled with “*1” apply only to TNS object files, items with “*2” apply only to accelerated object files, items with “*3” apply only to TNS/R native object files, and items with “*4” apply only to native SRL object files.

INFO SAVEFILE

The INFO SAVEFILE command displays information about save files. Save files can be located within the Guardian file system, or the OSS file system. The output displays new information related to SRLs and native processes.

Output

The INFO SAVEFILE command produces output of the following form:

```
Save File: savefile name

[                           Cause: COMMAND | ABEND | SIGNAL ]
                          Creator: CRUNCH | DMON | INSPECT
               Creation Timestamp: timestamp
                Creator's User ID: userid
                 Guardian Version: Lnn
[                     Trap Number: integer                  ]  *1
[                   Signal Number: integer                  ]  *1
[                     Wait Status: integer                  ]  *2
                        Processor: family
                     Program File: filename
         Program Binder Timestamp: timestamp
   Program Modification Timestamp: timestamp
                           System: name (number)
[             Truncated File Info: YES                      ]
[          Truncated Segment Info: YES                      ]

[                       Library Information                 ]
[                    Library File: filename                 ]
[         Library Binder Timestamp: timestamp               ]
[  Library Modification Timestamp: timestamp                ]
```

Library information will be repeated for each library used by the process. Items marked with “*1” are mutually exclusive (only one will be displayed) and will only be displayed if the process terminated due to a trap (or signal if a TNS/R native process), items marked with “*2” are for OSS processes only.

INFO SCOPE

The INFO SCOPE command includes information about the level the scope was compiled with. This will only be displayed for native code scopes. Items labelled with “*1” are only present for TNS/R native code scopes.

Output

```
Scope Name: name
Source File: file
Modification timestamp: timestamp
Compilation timestamp: timestamp
Language: lang
Type: CODE
Base: integer
Entry: integer
Length: integer words
[Optimization level: integer]        *1
```

INFO SIGNALS

The accepted syntax of the INFO SIGNALS command has not changed. The command will no longer be restricted to OSS processes, and the output will change to accommodate a 128-bit signal mask. The signal mask will be separated into four 32-bit numbers.

Output

The output template shows the information the INFO SIGNAL command displays without the DETAIL clause. Information on each signal is split across two lines.

```
-ETEST-INFO SIGNAL SIGFPE
Signal
  Handler            Mask                                Flags
SIGFPE(8)
  CRE_TRAP_HANDLER_    (0        0        0        0        ) 0
```

LIST PROGRAM

The accepted syntax of the LIST PROGRAM command has not changed, but the information displayed will. The detailed output will include the filename of all libraries used by this process.

Example

This example shows the previous output type.

```
-PROGRAM-LIST PROGRAM 1, DETAIL
Name: a.out
Number: 1
            General Information

              CPU,PIN: 8,160
[             OSS PID: 13406                          ]*1
     GUARDIAN Version: D20
      Instruction Set: TNS/R
             Location: #main.#5
            Processor: TNS/R (CLX)
      Program File: /usr/people/paul/bin/a.out
[           Libraries: /usr/lib/libc.a                 ]*2
[                      /usr/lib/libil8n.c              ]*2
                 State: HOLD
       System: \CUBS (175)
                  Type: TNS/R


          INSPECT Information

     ABEND Breakpoint: NO
     Code Breakpoints: 1
     Data Breakpoints: 0
        Source System: None
      STOP Breakpoint: NO
```

Fields labelled with a “*1” are listed only for OSS programs. Fields marked with a “*2” are listed only for programs that use libraries. A program may use zero or more libraries.

Usage Consideration

Location strings and program names that would extend beyond the last column are broken at the last column and listed in the next line beginning at the starting column of the Location field, or the Program Name field respectively.

MODIFY SIGNALS

The MODIFY SIGNALS command supports native NSK processes. Also, the action of SIG_DEBUG will be added to specify that when a signal is delivered, the debugger is to be invoked.

SELECT PROGRAM

The SELECT PROGRAM command allows multiple library files to be specified using the SRL clause.

SOURCE ICODE

The SOURCE ICODE command displays native RISC instructions. Inspect displays a minus (“-”) character next to RISC instructions which are from previous source lines and a plus (“+”) next to RISC instructions which are from subsequent lines. Lines containing RISC instructions also contain the source file line number that the instruction is for.

```
-PROG-SOURCE ICODE
   #3.1
   #4         int   global_1;
   #4.1       long  global_2;
   #4.2
1 *#5         main()
   #6         {
   #7           int   a;
   #8           short b;
   #9
         #5   00000000  BREAK INSPECT OCT
         #5   00000004  SW    ra,20(sp)

   #10      a = 13406544;
        #10        00000008  LUI t6,0xCC
        #10        0000000C  ORI   t6,t6,0x9150
     #10      00000010  SW    t6,28(sp)
   #11
    #12          printf("Hello World\n");
     #12      00000014  LUI a0,0x800
     #12       00000018  ADDIU a0,a0,0
       #12         0000001C  JAL   0x760E0940
     #12      00000020  NOP

  #13       if (errno)
        #13        00000024  LUI t7,0x5800
        #13        00000028  LW    t7,72(t7)
      #13      0000002C  NOP
       #13     00000030 BEQ  t7,$0,0x700003D0
      #13      00000034  NOP

 #13.1     b = 88;
        #13.1    00000038  LI    t8,88
     #13.1   000003C  SH    t8,26(sp)

 #14       }
       #14        00000040  OR    v0,$0,$0
        #14         00000044  BEQ   $0,$0,0x700003DC
      #14       00000048  NOP
     #14      0000004C  LW    ra,20(sp)
     #14      00000048  NOP
        #14         0000004C  LW    ra,20(sp)
        #14        00000050  NOP
        #14        00000054  JR    ra
      #14       00000058  ADDIU sp,sp,32
```

TRACE REGISTERS

The TRACE REGISTERS command displays TNS/R frames. For TNS/R frames, Inspect displays the PC register in the normal Inspect form as a scope name and offset. Inspect also displays the virtual frame pointer (VFP) for that frame.

Debugging at the TNS/R Native Machine Level

Inspect provides some functionality for examining the TNS/R machine state when debugging native programs. For more information about debugging at the TNS/R native machine level, see Section 15, Using Inspect on a TNS/R System, and the *Debug Manual*.

Examples

1. Translating Machine Addresses into Procedure names

   Inspect can be used to convert machine level TNS/R code addresses into procedure names. The following example displays the RISC $PC register as TYPE LOCATION. The output indicates that the current value of the $PC register is at procedure #M statement 2, line number 34 in source file \SYS.$VOL.SUBVOL.FILE, and at hexadecimal offset F.

```
-PROG-DISPLAY REGISTER $PC TYPE LOCATION
REGISTER $PC = #M.2, #M.#34(\SYS.$VOL.SUBVOL.FILE), #M + %HFI
```

   The following example uses an arbitrary code address instead of the $PC register.

```
-PROG-DISPLAY VALUE %H70000390 TYPE LOCATION
#A.1, #A.#23(\SYS.$VOL.SUBVOL.FILE), #A + %H0I
```

   This can be useful if you want to determine where a TNS/R JAL instruction will jump to.

```
-PROG-SOURCE ICODE FOR 1 STATEMENT
  *#38.1         CALL B (Def^1+1, Local^1);
          #38.1        00000080  LI    a0,11
          #38.1        00000084  JAL   0x70000560
          #38.1        00000088  ADDIU a1,sp,62

-PROG-DISPLAY VALUE %h70000560 TYPE LOCATION
 #B.1, #B.#75(\SYS.$VOL.SUBVOL.FILE), #B + %H0I
```

2. Compiling at Optimization Level 1

   The following source code has been compiled at optimization 1.

```
#28      PROC M MAIN;
#29      BEGIN
#30        INT     Local^1;
#31        INT     Local^2;
#32        INT     Local^3;
#32.01
#32.1      Local^1 := Lit^1 + 60;
#33        Local^1 := Local^1 / 2;
#34        CALL A (Def^1, Local^2);
#36        Local^1 := (Def^1*6) / 2;
#38        Local^3 := Local^1 + Local^2;
#38.1      CALL A (Def^1+1, Local^1);
#40       END;
```

   Inspect emits warning 198 at the first statement of the program, which indicates that an optimization has occurred on this statement. The SOURCE ICODE and INFO LOCATION commands can be used to determine the effects.

```
$VOL SUBVOL 61> RUND PROG
INSPECT - Symbolic Debugger - T9673D40 - (29JUL03)   System \SYS
Copyright Tandem Computers Incorporated 1983, 1985-1995
INSPECT
175,09,00207  PROG  #M.#33(FILE)
** Inspect warning 198 ** Results might be unexpected due to optimization.
Executing command file $VOL.SUBVOL.INSPCSTM

-PROG-SOURCE ICODE FOR 2 STATEMENTS
  *#33          Local^1 := Local^1 / 2;
         #33          00000000  LI    t7,80
         #33          00000000  LI    t7,80
         #28       -  00000004  ADDIU sp,sp,-64
         #32.1     -  00000008  LI    t6,160
         #28       -  0000000C  SW    ra,52(sp)
         #32.1     -  00000010  SH    t6,62(sp)
         #33          00000014  LI    at,0x8000
         #33          00000018  SLT   at,t7,at
         #33          0000001C  BNE   at,$0,0x70000314
         #33          00000020  NOP
         #33          00000024  BREAK BOUNDS
         #33          00000028  SLTI  at,t7,-32768
         #33          0000002C  BEQ   at,$0,0x70000324
         #33          00000030  NOP
         #33          00000034  BREAK BOUNDS
         #33          00000038  SH    t7,62(sp)

  #34          CALL A (Def^1, Local^2);
         #34          0000003C  LI    a0,10
         #34          00000040  JAL   0x70000220
         #34          00000044  ADDIU a1,sp,60
```

This example illustrates that two instructions have been moved into statement #33 from the prolog code at statement #28. The same is true for statement #32.1. The INFO LOCATION command can be used to display information about the statements of a procedure.

```
-PROGC-INFO LOCATION *
Scope: M

Compile File:  \SYS.$VOL.SUBVOL.FILE          Modified:
1995-08-03 17:13:42.00
                 Byte
Num  Line     Offset  Optimize Verb
1    #33      %0      Merged
2    #34      %74
3    #38      %110    Merged
4    #38.1    %200
5    #40      %214
```

This example illustrates that statements 1 and 3 have been affected by some type of compiler optimization. Inspect will emit warning 198 whenever it encounters one of these statements.

Contrast the previous output with the same procedure compiled at optimization 0.

```
-EPTAL2-INFO LOCATION *
Scope: M

Compile File:  \SYS.$VOL.SUBVOL.FILE         Modified:
1995-08-03 17:13:42.00
                Byte
Num  Line      Offset  Optimize Verb
1    #28       %0
2    #32.1      %10
3    #33       %20
4    #34       %120
5    #36       %140
6    #38       %150
7    #38.1      %240
8      #40          %260
```

Statements #28, #32.1, and #36 have not been combined with neighboring statements.

3. Support of Register-based Variables

   The following source compiled at optimization 1, illustrates how Inspect supports register-based variables.

```
#16   PROC A( Param^1, Param^2);
#17     INT   Param^1;
#18     INT  .Param^2;
#19    BEGIN
#20     INT   Local^1;
#21
#22     Local^1 := 15;
#23     IF Param^1 = Def^1 THEN
#24       Param^2 := Lit^1
#25     ELSE IF Param^1 = Def^1 + 1 THEN
#25.1      BEGIN
#26        Local^1 := Local^1 - 15;
#26.001     Param^2 := Param^2 * 31000;
#26.01     END;
#26.1   END;
```

The INFO IDENTIFIER command provides information about a symbol.

```
-PROG-INFO IDENTIFIER Param^1
PARAM^1: VARIABLE
storage^info:
TYPE=BIN SIGN, ELEMENT LEN=16 BITS, UNIT SIZE=1
ELEMENTS
access^info:
REGISTER 4 ($A0)

-PROG-INFO IDENTIFIER Param^2
PARAM^2: VARIABLE
storage^info:
TYPE=BIN SIGN, ELEMENT LEN=16 BITS, UNIT SIZE=1
ELEMENTS         access^info:
REGISTER 5 ($A1)X
```

The output above illustrates that parameter Param^1 is stored in register $4 or $A0 and that parameter Param^2 is stored in register $5 or $A1 and is indirect. The Inspect DISPLAY and DISPLAY REGISTER commands can be used to display the contents of these variables.

```
-PROG-DISPLAY Param^1
PARAM^1 = 10
-PROG-DISPLAY REGISTER $4
REGISTER $4 = 10

-PROG-DISPLAY Param^2
PARAM^2 = 99
-PROG-DISPLAY REGISTER $5
REGISTER $5 = 1342177068
-PROG-DISPLAY (1342177068) TYPE INT
99
```

4. Using a Signal Handler to Control Program Flow

   Inspect allows you to change a signal handler to several values, including SIG_DEBUG which can be used to gain control of a program when a signal is delivered. For example, if a TNS/R native application is receiving an arithmetic overflow trap (signal SIGFPE), change the signal handler for SIGFPE and rerun the program. For example:

```
-PROG-MODIFY SIGNAL SIGFPE
Signal SIGFPE(8)
Handler = SIG_DFL := SIG_DEBUG
Mask = 0 0 0 0 :=
Flags = 0 :=

-PROG-INFO SIGNAL SIGFPE, DETAIL

                 Signal:SIGFPE(8)
                Handler:SIG_DEBUG
                   Mask:0 0 0 0
                  Flags:0

-PROG-RESUME
INSPECT  SIGNAL %10 - Entered debug due to non
deferrable signal
241,00,00274 PROG #A.#26.001(FILE) + %H10I
-PROG-TRACE
Num Lang Location
 0  TAL #A.#26.001(FILE) + %HFI
 1  TAL #M.#40(FILE)
-PROG-INFO SIGNAL %10, DETAIL

               Signal:SIGFPE(8)
              Handler:SIG_DEBUG
                 Mask:0 0 0 0
                Flags:0
```

Statement 26.001 in procedure A is the location of the arithmetic overflow.

18 Using Inspect on a TNS/E System

Capabilities of Inspect on TNS/E Systems

What Inspect Does

On a TNS/E system, you can use Inspect to debug the following:

- TNS processes emulated on a TNS/E system. Emulated TNS processes are TNS processes that either:
  - Were accelerated beforehand with the TNS Object Code Accelerator (OCA) and are running in TNS accelerated mode
  - Are being interpreted and are running in TNS interpreted mode
- Snapshots of emulated TNS processes generated on either a TNS/R or TNS/E system
- Snapshots of TNS/R native processes

What Inspect Does Not Do

Inspect cannot be used to debug TNS/E native processes or TNS/E native snapshots on a TNS/E system.

New Features

No new features have been added to Inspect for deployment on TNS/E systems.

Inspect behaves essentially the same on a TNS/E system as on a TNS/R system, with a few exceptions and restrictions as noted in Table 18-2 on page 18-8. Restrictions are also described in the individual command descriptions in Section 6, High-Level Inspect Commands.

Languages Supported

Programming languages supported by Inspect on TNS/E systems include TNS Fortran, Screen COBOL, TNS COBOL85, Pascal, TAL, and TNS C/C++.

Acceleration on TNS/E Systems

TNS program files that have been accelerated with the TNS Object Code Accelerator (OCA) run in TNS accelerated mode on TNS/E systems. If a TNS program file has not been accelerated, it will run in TNS interpreted mode. These two execution modes are further described in the following paragraphs.

TNS Accelerated Mode

The TNS Object Code Accelerator (OCA) translates TNS instructions to equivalent Itanium instructions. OCA augments a type-100 TNS object file with a new region containing the Itanium instructions (the Itanium region, shown in Figure 18-1, Acceleration of TNS Object Code on TNS/E Systems, on page 18-3).

TNS object files that have been optimized by OCA are called accelerated object files, or accelerated program files if they include a main procedure. Programs that have been accelerated run in TNS accelerated mode, which is significantly faster than TNS interpreted mode. You must apply OCA once to the object file before run time, or the TNS program runs in TNS interpreted mode, which is considerably slower.

TNS Interpreted Mode

In TNS interpreted mode, individual TNS 16-bit stack machine instructions are repeatedly decoded at run time and simulated one at a time, using only the in-memory image of TNS code segments (ignoring any optional file augmentation).

TNS interpreted mode interoperates with TNS accelerated mode, in which entire programs or libraries have been previously translated into equivalent but optimized sequences of TNS/E machine instructions.

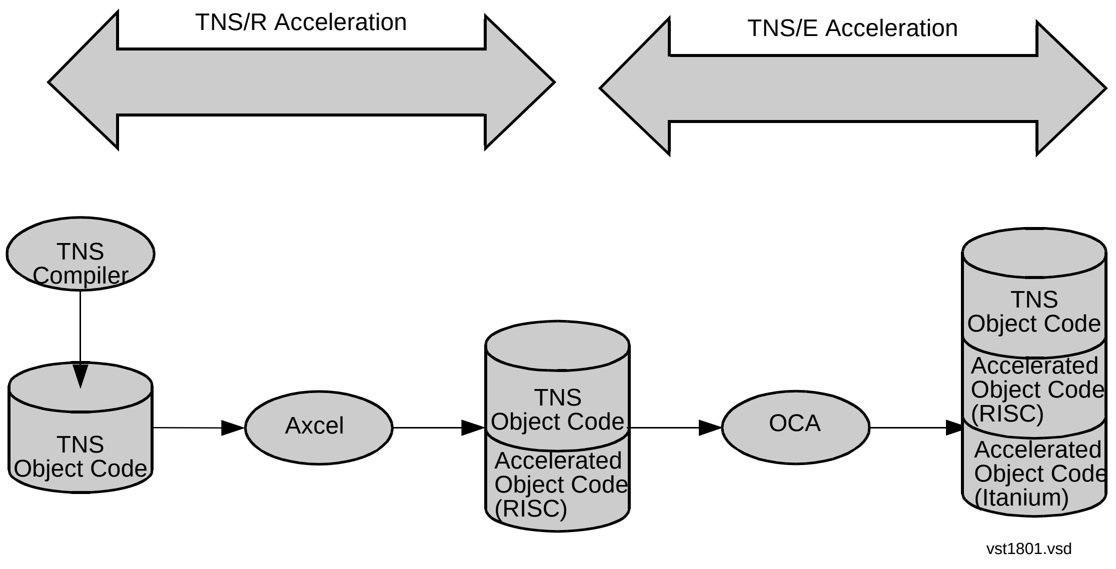
Programs switch automatically between execution modes when branching from untranslated object files to OCA-augmented object files, and vice versa. For example, a program file might run in accelerated mode but a user library might be interpreted.

Interpreted TNS mode is used when executing TNS object files that could not be accelerated. It might also be used for brief periods with accelerated programs, when the TNS code sequence could not be translated.

Accelerating TNS Processes

Figure 18-1 illustrates acceleration on a TNS/E system by both the TNS Object Code Accelerator (OCA) and the TNS/R accelerator, Axcel.

Figure 18-1. Acceleration of TNS Object Code on TNS/E Systems



You can run OCA on a TNS object file that has already been accelerated using the TNS/R accelerator (Axcel).

You can also accelerate a TNS object file using both OCA and Axcel in any order. For example, you can run OCA on a TNS object file that has previously been accelerated for a TNS/R system using the Axcel accelerator. And you can run Axcel on a TNS object file that has been previously accelerated for a TNS/E system using OCA.

The result of this double acceleration (shown in Figure 18-1) is a doubly augmented object file that contains three code areas:

- TNS object code generated by a TNS compiler
- Accelerated (RISC) object code generated by Axcel
- Accelerated Itanium object code generated by OCA

A doubly augmented file can run in accelerated mode on both TNS/R and TNS/E machines.

For more information about using OCA and about running and debugging accelerated programs, see:

- *H-Series Application Migration Guide*
- *Object Code Accelerator (OCA) Manual*

Debugger Selection on TNS/E Systems

Debugger selection is determined by the debugging attributes of a process. The rules for debugger selection differ according to system type (TNS/R or TNS/E). The rules followed on a TNS/R system are different from the rules on a TNS/E system. In addition, the TNS/R system debugger is Debug, but the system debugger on TNS/E systems is Native Inspect.

When a debugger is invoked on a TNS/E system, the system software selects the debugger according to the rules of precedence described in Table 18-1.

Table 18-1. Debugger Precedence on TNS/E Systems

Process Type	INSPECT Attribute	Debugger Precedence
TNS	INSPECT ON	Visual Inspect, Inspect, Native Inspect
TNS	INSPECT OFF	Inspect, Native Inspect
TNS/E Native	INSPECT ON	Visual Inspect, Native Inspect
TNS/E Native	INSPECT OFF	Native Inspect

Considerations About Debugger Selection

- Visual Inspect can only be selected when you have already established a matching Visual Inspect client connection.
- If the Inspect subsystem is not able to find a suitable debugger (either Visual Inspect or Inspect), the debugger selected is the TNS/E system debugger, Native Inspect, which is not part of the Inspect subsystem.
- Native Inspect has extremely limited support for debugging TNS processes. Therefore, Inspect is the preferred debugger for TNS processes even if INSPECT is set to OFF.
- COBOL programs on a TNS/E system must be debugged using either:
  - Inspect for TNS COBOL85 programs
  - Visual Inspect for either TNS COBOL85 or TNS/E COBOL programs

Table 18-1 illustrates that on a TNS/E system, Inspect is invoked as the chosen debugger to debug a TNS emulated process in only two cases -- when the INSPECT attribute is ON and Visual Inspect is not available, or when INSPECT is OFF (also see Figure 18-2 on page 18-6).

For background information about the debugging attributes of a process, see The Debugging Attributes of a Process on page 4-4.

Debugger Selection Criteria

In the flowcharts shown in both Figure 18-2 on page 18-6, and Figure 18-3 on page 18-7, debugger selection criteria are defined:

Criteria	**Meaning**
INSPECT attribute on?	The setting for the INSPECT attribute is set ON for the process you will debug (set with TACL, the compiler, or the linker).
Visual Inspect session?	You have started Visual Inspect and have connected to the NonStop host on which the process to be debugged will run. The user ID of the process must match the user ID that was used to log on to Visual Inspect.
Inspect available?	The Inspect subsystem (IMON, DMON, $DM*nn*) is running, and the Inspect command-line interface is available.

Figure 18-2 illustrates the debugger selection process for an emulated TNS process on a TNS/E system. Figure 18-2 demonstrates that to debug a TNS process on a TNS/E system, you can use either Visual Inspect (the preferred debugger) or Inspect.

Note that Native Inspect can in some cases be selected as the debugger for a TNS process on TNS/E. Native Inspect provides very limited debugging for TNS processes, but can provide views of the TNS/E instructions and registers for TNS processes accelerated with OCA. In addition, only Inspect or Visual Inspect can be used to debug COBOL programs on a TNS/E system.

Figure 18-2. Debugger Selection for a TNS Process on a TNS/E System

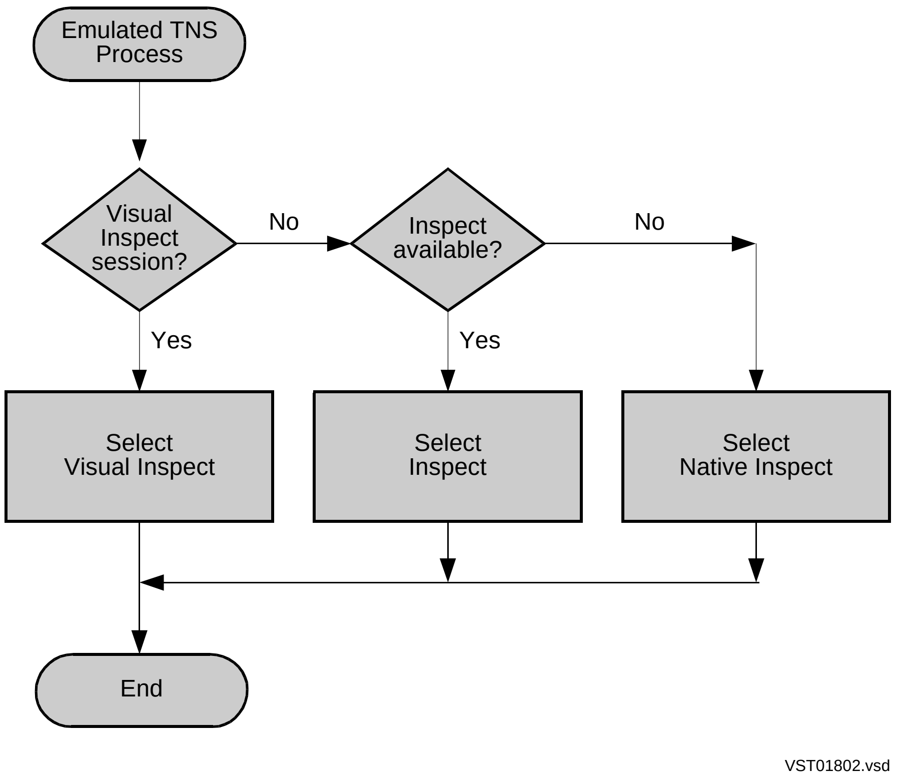
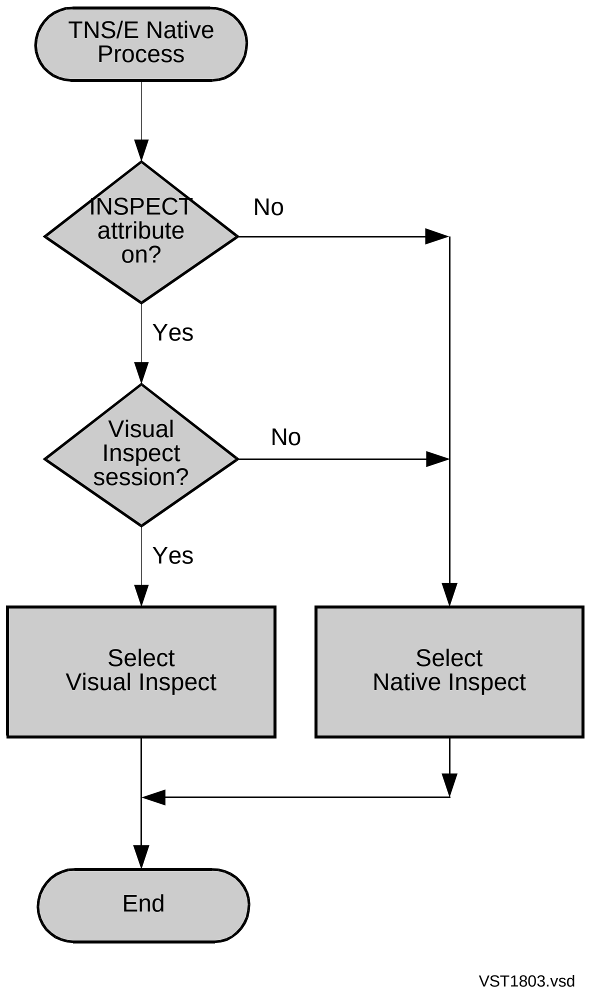


Figure 18-3 illustrates the debugger selection process for a native TNS/E program file on a TNS/E system.

Note that to debug a TNS/E native process, you must use either Visual Inspect or Native Inspect; you cannot use Inspect to debug a TNS/E native process. In addition, only Inspect or Visual Inspect can be used to debug COBOL programs on a TNS/E system.

Figure 18-3. Debugger Selection for a TNS/E Native Process



Using Inspect to Debug TNS Programs on TNS/E Systems

Debugging TNS programs on TNS/E systems is virtually the same as debugging TNS programs on TNS/R systems. The capabilities of Inspect are the same on the two platforms, except that a few Inspect commands have the limitations listed in Table 18-2.

Much of the information in Section 16, Using Inspect With Accelerated Programs on TNS/R Systems also applies to debugging TNS programs on TNS/E systems, with the exceptions noted in Table 18-2.

In general, descriptions throughout this manual that pertain to debugging accelerated programs on TNS/R systems also apply to debugging TNS programs on TNS/E systems.

Table 18-2. Considerations for Inspect Commands on TNS/E Systems

Command	Limitation on TNS/E Systems
ADD PROGRAM	You cannot add a TNS/E process or a TNS/E snapshot to an Inspect debugging session on a TNS/E system.
DISPLAY REGISTER BOTH	TNS/E registers cannot be viewed using Inspect. Inspect displays only the TNS registers on TNS/E systems.
ICODE	The ICODE output for an OCA process will show safe-point annotations, but the ICODE output for a snapshot of an OCA process will not show safe-point annotations. There is no display of TNS/E ICODE.
ICODE BOTH	Inspect does not display Itanium processor instruction codes.
INFO OBJECTFILE	Inspect does not read TNS/E native object files.
INFO SAVEFILE	Inspect does not read TNS/E snapshot files.
SOURCE	Shows register-exact point and deleted statement annotations for an OCA process but not for an OCA snapshot.
TRACE	Does not show any information about TNS/E native frames.

A Error and Warning Messages

Inspect scans command lists command by command and notifies you of an error or potential error by issuing an error or warning message:

- An error message indicates that Inspect could not execute the command. When Inspect generates an error message, it disregards the command that caused the error and any commands that followed it in the command list.
- A warning message indicates that Inspect was able to execute the command, but that the user should be made aware of some detected condition. When Inspect generates a warning message, it completes the command that caused the warning and continues to the next command in the command list.

Note. The file INSPMSG must be located in the same volume and subvolume as Inspect in order for Inspect to access the textual messages it contains. If this file cannot be located, Inspect issues only a number associated with a particular error or warning. No message appears to help you diagnose the cause of the error.

Fatal Errors During Session Start-Up

If IMON encounters a system error that prohibits it from creating an Inspect process, it reports a fatal error and does not start an Inspect session. For example, if the intended swap volume for the Inspect process is full, the operating system reports a system error to IMON; IMON then reports a fatal error.

If Inspect encounters a fatal error, it will issue an internal stack trace and then ABEND. On TNS/R systems, you must stop any processes you were debugging. On TNS/E systems, the process will be activated. Report any fatal errors to your HP representative.

HELP Availability

Error and warning message descriptions are available in online help. To access the message descriptions from online help, enter “HELP ERROR” followed by the error or warning number.

1

```
Invalid syntax
```

The sequence of input characters does not result in a valid Inspect command. Inspect redisplays the input and places a caret (^) under the point where it detected the error.

2

```
Unterminated continuation line
```

While scanning for the remainder of a continuation line, a line following a line ending with a “&,” Inspect encountered the end-of-file. This generally occurs only when Inspect is reading commands from INSPLOCL, INSPCSTM, or an OBEY file. However, Inspect also generates this message if you press CTRL/Y when entering a continuation line.

3

```
Registers R0-R7 and E are not valid in the STOP state
```

This is a warning. You attempted to display registers in a program while in the STOP/ABEND state.

4

```
Effective input record is too long
```

The command list exceeded 512 characters after Inspect expanded any aliases it contained.

5

```
Integer conversion error
```

The command list specified an invalid integer.

6

```
Unterminated string
```

A string specified in the command list is missing its closing delimiter (apostrophe or quotation mark).

7

```
Identifier too long
```

The command list included an identifier longer than the Inspect maximum of 31 characters.

8

```
Invalid file name file-name
```

The given file name does not conform to syntax conventions.

9

```
Invalid Subvolume name
```

A volume or subvolume name specified in the command list is too long (longer than eight characters), contains an invalid character, or specifies a volume that does not exist.

10

```
Invalid System name
```

A system name specified in the command list is too long, contains an invalid character, or specifies a system that does not exist.

11

```
Numeric range exceeded
```

A number specified in the command exceeds the allowed range. For example, there is a well-defined limit for an input value used with a TEMP clause, bit-field or segment ID.

12

```
OBEY nesting exceeds maximum
```

The OBEY command would cause nesting of OBEY commands beyond four levels, which is the maximum Inspect supports.

13

```
Registers are not valid in the STOP state
```

This is a warning. You attempted to display registers in a program while in the STOP state.

14

```
Invalid OBEY file - ignored
```

The OBEY file specified in the command list is already in use as an input file or has a file code of 100.

15

```
Invalid LOG file - ignored
```

The LOG file specified in the command list is already in use as an input file or as an OUT file.

16

```
Invalid RADIX value
```

The SET RADIX command specified a radix other than DECIMAL, HEXADECIMAL, or OCTAL.

17

```
Starting Location is not a memory-exact point:
listing starts at next memory-exact point
```

This is a warning. When the location displayed is not a memory-exact point, Inspect rounds the starting location to the next memory-exact point.

18

```
Operation allowed on OSS systems only
```

You attempted a command on a operation valid only on OSS systems.

19

```
File is not an object file: file-name
```

The specified object file does not have the file code 100, which signifies a program file.

21

```
Invalid operation on saved program
```

The current program is a save file, and the requested operation is not allowed on a save file. When examining save files, you can display data, but you cannot modify data or use execution control commands.

22

```
Invalid address
```

An address specified in the command list is not a valid address in the current program.

23

```
All TNS registers are out of date
```

This is a warning. You attempted to display a specific register in a program while in the STOP/ABEND state.

24

```
Non-existent program
```

The command list specified a process that is no longer under the control of the current Inspect session.

25

```
Security error
```

The requested operation is not allowed for security reasons. For example, data and registers may not be modified while a process is executing system code.

26

```
Invalid data address
```

A data address specified in the command list is not a valid data address in the current program.

27

```
Invalid code address
```

A code address specified in the command list is not a valid code address in the current program.

28

```
Request invalid in current state
```

The requested operation is not valid given the execution state of the current program. For example, the MODIFY command is invalid when the current program is in the run state.

29

```
TCP internal error
```

An internal error occurred on a TCP process.

30

```
DMON not open
```

The Inspect process is unable to communicate with the DMON process in the processor executing the requested program. You should end the current Inspect session and start a new one.

31

```
TCP not open.
```

Inspect is unable to communicate with the TCP process controlling with the requested SCREEN COBOL program. You should end the current Inspect session and start a new one.

32

```
Program not found
```

The command list specified a process that has stopped, or the program name is not properly specified.

33

```
Operation not supported for SCOBOL
```

The requested operation is not supported for SCREEN COBOL programs.

34

```
Invalid operation on a NonStop I+ system
```

The requested operation is not supported on a NonStop 1+ system.

35

```
Proc undefined: scope-unit
```

Inspect could not find the specified scope unit in the current program.

36

```
No symbols available in scope: scope-unit
```

The program file does not include symbols for the specified scope unit.

38

```
Breakpoint already set
```

A breakpoint already exists at the specified break location. When using low-level Inspect, you must clear a breakpoint before you reset it.

39

```
Address out of bounds
```

The address specified is not in the range of valid addresses in the segment.

40

```
Segment not allocated
```

An extended data segment specified in the command list has not been allocated by the current program.

41

```
Inspect not enabled for process
```

The process specified for debugging does not have the Inspect attribute set.

42

```
Required DMON not active
```

The BREAK command specified the BACKUP clause, and the CPU executing the backup process has no active DMON process. Contact your system manager or system operator.

43

```
No backup process active
```

The BREAK command included the BACKUP option, but the current program has no backup process.

44

```
No active program
```

The command specified requires an active program.

45

```
Invalid request
```

The requested operation is not valid on the current program, possibly due to the type of the program or its state.

46

```
Breakpoint table full
```

Inspect cannot create the specified breakpoint because the breakpoint table for the CPU or TCP executing the current program is full. For CPUs, the maximum number of breakpoints is set at system generation. For TCPs, the maximum number of breakpoints is 20.

47

```
Terminal must be a Guardian device
```

The specified argument to the TERM command is not a Guardian device.

48

```
Requested file is not available file-name
```

The file specified in the DELETE SOURCE OPEN command refers to a source file that is not open. Use the LIST SOURCE OPEN command to display the list of open source files.

49

```
Timestamp mismatch for file-name
```

Following this warning, Inspect displays the current timestamp and the recorded timestamp. Both program files and source files can generate this warning:

- For program files, this warning indicates that the file's timestamp differs from the timestamp recorded in the save file. Note that modifications to the program file might invalidate the correspondence between the symbol information in the program file and references to symbols in the save file.
- For source files, this warning indicates that the file's timestamp differs from the timestamp recorded in the program file. Note that modifications to the source file might invalidate the correspondence between code locations and source lines.

50

```
Operation allowed only on OSS programs
```

This error is emitted for commands entered on Guardian programs but intended for OSS programs only, such as MODIFY or INFO SIGNAL.

51

```
Format string invalid
```

If you use DISPLAY FORMAT in a command list, it must be the last command in the list because Inspect interprets all text following the FORMAT (or FMT) keyword as the format list.

52

```
Format data buffer overflow
```

The DISPLAY FORMAT specified a format longer than 24 lines, which is the maximum that the FORMAT clause can display.

53

```
No such instance of procedure on stack: scope-unit
```

The command list specified a nonexistent instance of the given scope unit.

54

```
No help available for topic
```

There is no help information available for the given topic.

55

```
File is not a terminal
```

A device other than a terminal was specified when a terminal was expected.

56

```
Invalid TERM command - ignored
```

The TERM command was issued from a non-interactive source, such as an OBEY file. The TERM command is valid only when entered interactively.

57

```
Unknown signal: signal id
```

The specified signal is not a valid signal type.

58

```
Value must be in the range 1 to 32767 inclusive
```

You specified an incorrect range with the FOR clause of the SOURCE command. A range of 0 or greater than 32767 is not valid.

59

```
FORMATDATA conversion error: error-number
```

The FORMATDATA procedure returned the reported error number. Refer to the manual for interpretation of the error number.

60

```
Expression evaluation error
```

An expression that in the command list contained an unknown identifier or caused an overflow or type conflict.

61

```
Undefined variable: identifier
```

Inspect could not find the specified identifier.

62

```
Save file code not 130: file-name
```

The file specified for use as a save file does not have the file code 130, which signifies a save file.

63

```
Modification is not allowed while in system code
```

The current program has been held at a system code location. For security reasons, no data modification is allowed. A breakpoint may be placed at the first user code location on the stack. When this breakpoint is reached, then the MODIFY command may be reentered.

64

```
Invalid target symbol for modify
```

The MODIFY command specified a data location to be modified which cannot be modified.

65

```
Not a disk file: file-name
```

The specified file was expected to be a disk device.

66

```
Too many values for modify variable
```

The specified target data location in the MODIFY command does not contain as many elements as are specified in the modify item list. Note that a subscript range may be specified on the data location, indicating which elements of an array are to be modified.

67

```
Modify location must be element level
```

The item specified as the target of a MODIFY command is a group-level item. Element level items must be specified, or the WHOLE option used.

68

```
Invalid operation in stop state
```

The current program is in the stop state. The requested operation is not allowed in the stop state.

69

```
Invalid operation in run state
```

The current program is in the run state. The requested operation is not allowed in the run state. Use the HOLD command to put the program into the hold state.

70

```
Specified segment can not be selected
```

You attempted to select an unselectable data segment.

72

```
Inactive program unit reference may not contain offset
```

For an inactive SCREEN COBOL program unit, Inspect will allow only references to scope names with no offset and no paragraph or section name attached. A breakpoint must be set at the beginning of the scope. When that breakpoint is reached, then breakpoints may be set at offsets within the program unit.

73

```
Offset specified exceeds locations in scope unit
```

A code offset specified results in an address outside the scope unit. Note that in high-level Inspect the default radix is decimal and the default code unit is STATEMENTS.

74

```
Paging file error:  file error message(nnn)
                    ALLOCATESEGMENT error code nnn
```

This error might result when there is insufficient disk space for Inspect to allocate its extended data segment; Inspect uses the same volume as the swap volume of the initial process that requested Inspect services. Refer to the *Guardian Procedure Errors and Messages Manual*.

75

```
File System Error
```

A File System Error has occurred. Inspect will issue this error followed by the number.

76

```
Invalid value for variable size item: identifier
```

The identifier reported has a variable size which evaluates to an invalid value.

77

```
Invalid value for variable dimension: identifier
```

The identifier reported was declared with one or more variables as its dimensions. One of these variables is undefined or evaluated to an invalid value.

78

```
Invalid subscript value(s) for variable: identifier
```

An identifier in a subscript expression is undefined or a specified lower bound is greater than a specified upper bound.

79

```
Subscript value outside of declared bounds: identifier
```

The specified identifier was declared to have dimension bounds, but the subscript value specified is outside these bounds.

80

```
Required subscript missing: identifier
```

The specified identifier was declared to be used with subscripts. Subscripts must be added to the identifier reference in the command.

81

```
Too many subscripts for variable: identifier
```

The specified identifier was declared to be used with fewer subscripts than appear in the command. One or more of the subscripts should be removed.

82

```
Incompatible IMON version: version-code
```

The IMON process is of an incompatible version code. The number printed is the version code. An Inspect version upgrade may be required.

83

```
Incompatible DMON version: version-code
```

The DMON process in the CPU executing the program is of an incompatible version code. The number printed is the version code. An Inspect version upgrade may be required.

84

```
Incompatible TCP version: version-code
```

The TCP process is an incompatible version code. The number printed is the version code. An Inspect version upgrade may be required.

85

```
Variable is local to inactive scope: identifier
```

The specified variable is local to a scope unit which is not currently active. No instance of the scope unit appears in the call history. The TRACE command displays active scope units.

87

```
Maximum lines exceeded for format output
```

A warning. The maximum number of lines which may be produced by a single DISPLAY command with the FORMAT option is 24. This maximum was exceeded.

88

```
Invalid object file: file-name
```

The object file specified contains conflicting internal data.

89

```
Invalid copy value on modify
```

An invalid COPIES value was specified on the MODIFY command. The value must not exceed 32767.

90

```
Incompatible Save File version: version-code
```

The save file specified contains an incompatible version code. The number printed is the snapshot file version code. An Inspect version upgrade may be required.

91

```
Incompatible Inspect version
```

The DMON process has detected an incompatible version code. A DMON version upgrade may be required.

92

```
Display item is truncated
```

A warning. The item being displayed has been truncated to 1014 bytes for the display. The WHOLE option may be used to display the entire item.

93

```
Specified location is a data address
```

The requested display will show instructions, but you entered a data location instead of a code location.

94

```
Maximum constant length (238 bytes) is exceeded.  Value is
truncated.
```

A warning. The maximum length constant (string) allowed is 238 bytes. Characters beyond this length have been truncated.

95

```
Maximum string length (250 bytes) is exceeded.  String is
truncated.
```

A warning. The maximum length string allowed is 250 bytes. Characters beyond this length have been truncated.

96

```
Previous breakpoint is replaced.
```

A warning. The BREAK command specified a breakpoint requiring replacement of a previously set breakpoint.

98

```
Qualification required to resolve ambiguous reference:
identifier
```

The specified identifier must be qualified further to distinguish it from other identifiers of the same name.

99

```
Access is local (sublocal reference must be qualified):
identifier
```

A warning. The specified TAL variable is declared as a local in the current procedure and as a sublocal in the current subprocedure. The local variable is used. If access to the sublocal is desired then the identifier must be qualified with the subprocedure name.

100

```
Subscript range is not allowed in MODIFY WHOLE target
variable.
```

The MODIFY command with the WHOLE option cannot specify a target data location with a subscript range. Each element of the array must be modified by a separate command.

101

```
No commands can follow fix commands, RESUME, or STEP in a
command list
```

The command preceding the indicator must be the last command in a command list. The commands this applies to are RESUME, STEP, and interactive modify.

102

```
'F' option is valid only for Fortran programs
```

The INFO OPENS command specified the F option, but the current program does not contain FORTRAN scope units.

103

```
Modify string must be provided
```

The MODIFY command with a WHOLE option may not be interactive. The modification string must be included in the command.

104

```
TCP backup takeover.  All requestor breakpoints must be
reentered.
```

The TCP process for the current PATHWAY requester program has stopped and its backup has taken over. All breakpoints in the program have been lost and must be reset.

105

```
DMON communication lost.  Program is deleted: program-file
```

The listed program is removed from the Inspect program list. If the program was the only program being debugged, then the Inspect process will stop. A likely cause of this error is that the DMON process has been stopped or the processor has halted. If the processor has not halted, then the program being debugged may still be running.

106

```
Invalid numeric digit for current input base
```

The indicated digit is not valid in the current input base.

107

```
Break parameter is repeated in command
```

BREAK command parameters may not be repeated in the command. The command is ignored.

108

```
Integer to string conversion required
```

A warning. The operation required a conversion from an integer to string.

109

```
Non-numeric character in numeric string
```

A warning. A numeric string variable being displayed contains nonnumeric characters. The displayed value is probably incorrect.

111

```
Scope undefined: scope-unit
```

Inspect does not recognize the specified scope at the current location in the program. For programs that are not SCREEN COBOL programs, you can use the MATCH SCOPE command to list the scope names that match a given pattern.

113

```
Privilege is required for specified access
```

You made a request that you do not have the authority to make. For example, nonprivileged users cannot display or modify system data.

114

```
Super-super id is required for command
```

Certain Inspect commands, such as the SET PRIV MODE command, require that the Inspect process have the user.ID 255,255.

115

```
Data not accessible in a saved program
```

In a saved program, some portions of the environment are not available. You accessed one of these portions.

116

```
Invalid line number
```

The line number you entered is not a valid line number.

117

```
A subsequent line number is assumed: line-number
```

A warning. A line number specified as a code-location or data-location in a command had no executable statement associated with it. The next executable statement begins at line number line-number.

118

```
Line number is not recognized, try specifying source file
also:  line-number
```

Inspect does not recognize the given line number for one of these reasons:

1. It is less than the first line number defined in the scope unit.
2. It is greater than the last line number defined in the scope unit.
3. It is in a source file other than the one containing the declaration of the scope unit.

120

```
Source file is not recognized: file-name
```

Inspect cannot find a source file that you specified in the command list either as part of a code location or as the parameter to DELETE SOURCE OPEN.

121

```
Fatal Trap trap-number AT P = %nnnnnn IN space-id
```

Inspect has encountered a trap at the given location; a stack trace usually follows this message. Contact your Tandem representative.

122

```
Specified proc refers to a different code segment: space-id
```

For multiple code segment programs in low-level Inspect, you must indicate the code segment in addition to the address for the scope unit. If the scope unit is not in the code segment you specified, Inspect displays this error.

123

```
Unexpected frame data was encountered; stack is probably
corrupt
```

Inspect displays this warning message when you enter a TRACE command and Inspect detects that a stack marker contains an impossible L register. Inspect can detect when a stack is corrupt in a TNS program.

124

```
Line no longer exists: line-number
```

Inspect displays this message when you make a reference to a nonexistent line number. The line number might be nonexistent because you modified the source file during an edit session prior to debugging.

125

```
Not an Edit file: file-name
```

Inspect issues this error if you give the SOURCE command a scope unit name that does not refer to a source file. A scope unit refers to the source file from which it is compiled. If you change this source file to another type of file, say an object file, Inspect displays this error.

126

```
End-of-file on: file-name
```

Inspect displays this warning when you are using the SOURCE command to repeatedly display blocks of source lines by hitting the RETURN key, and you reach the end of the source file.

127

```
Breakpoint will occur upon access to containing word
```

This warning is reported when you set a write or read/write breakpoint on a variable that is less than 16 bits in size. When this is the case, memory access breakpoints will be reported when the variable is accessed in addition to when the portions of the word not containing the variable are accessed.

128

```
Maximum table size exceeded. Table size truncated.
```

In low-level Inspect, a display command using the table format exceeded the maximum table size for the Inspect message buffer. Inspect truncated the table to the maximum size.

129

```
New object file format, symbols cannot be accessed.
```

The symbols cannot be accessed because Inspect detected an inconsistency in the Inspect region of the object file. The object file was compiled with a compiler from a release that is more recent than the release of the Inspect you are using (for example, a C10 compiler and a C00 version of Inspect). Inspect defaults to the low-level mode for this session. Use Inspect from the same release as the compiler used to compile the object file.

130

```
Following value was too large, INSPECT cannot access
enumeration identifier
```

An attempt was made to display an enumerated type for which the literal corresponding to its value was not saved. The number of enumerated type literals stored in older object file versions was limited.

131

```
Volume name is too long to be used with a system
specification
```

Inspect encountered an invalid network volume name in a SOURCE command when a SOURCE SYSTEM command was in effect. Inspect cannot retrieve the source file from the remote system.

132

```
Program has no active user library; Library file will
be used for static symbol access only
```

This is a warning. When a program has no active user library and you want to select that program as the current program, the above warning is displayed.

133

```
Location is in an inactive user library
```

If you specify a user library to use, as in SELECT PROGRAM, and the program does not have a user library, subsequent commands will be rejected with the above error if they involve locations from the new user library.

134

```
Location not contained in valid code block
```

This error is reported when Inspect attempts to look up a procedure name and finds that BINDER information has been stripped from the object file.

136

```
No breakpoint number is available.
```

All available breakpoint numbers have been used. Inspect supports up to 99 breakpoints in each program. Remove any unneeded breakpoints to make more numbers available.

137

```
Save files created prior to B00 not supported
```

You used the ADD PROGRAM command to add a save file that was created prior to B00.

138

```
The file name cannot be converted into internal format
```

This error is reported when a file name cannot be converted into an internal format.

139

```
Invalid PROCESS HANDLE data format
```

This error is reported when given data is not a valid PROCESS HANDLE. An otherwise valid PROCESS HANDLE is invalid if it refers to a named process that is no longer running.

140

```
Incompatible run-time environment.  Environment is mode
```

This warning is reported when the environment mode in a program block is not compatible with the current runtime environment. mode is one of NEUTRAL, COMMON, HISTORIC, or UNKNOWN.

141

```
Common Run-time error number
```

This error is reported when Inspect detects a common runtime environment error.

142

```
Variable is read-only; breakpoint is set for read access
```

This warning is reported when you enter an invalid data breakpoint type on a read-only array. The data breakpoint type must be read. It cannot be change, write, or access.

143

```
Number cannot be represented in 64 bits
```

You attempted to enter an integer that cannot be represented in 64 bits. Enter an integer in the range of -9,223,372,036,854,775,808 through +9,223,372,036,854,775,807.

144

```
Current location is not within any procedure in any object
file associated with your program.
```

This warning is reported when Inspect attempts to associate a location with a procedure in an object file associated with a process or save file.

145

```
There is no scope associated with this location
```

This warning is emitted when Inspect attempts to associate a location, either from the operating system or found within a save file, when a procedure in some object file associated with the process or save file. The warning message is followed by output which identifies the location as an octal word offset in some code segment.

146

```
ACCESS is not allowed with READ or WRITE
```

This error is reported when you enter a break command with an illegal combination of data breakpoint types. ACCESS specifies a break event should occur on read access, in addition to write access of the data item.

169

```
Following value was invalid for enumerated type
```

An attempt was made to display an enumerated type that contains a value for which there is no corresponding enumeration. The program might have accidentally overwritten a variable with a value outside of the enumeration range.

170

```
Only pointers may be dereferenced: item
```

A dereferencing operator was applied to the item indicated by item, but item is not a pointer.

171

```
Value of tag field has no associated variant item
```

The value indicated was found in the tag field of a Pascal variant record, but it does not have a variant defined for it. Your program might have accidentally overwritten the tag field.

172

```
Step out maximum is 10
```

A STEP OUT command was issued with a count greater than 10. Inspect allows a maximum of 10 procedure levels to be stepped with one command. If you want to step more than 10 levels, issue several STEP OUT commands.

173

```
Step out is not allowed from the main scope unit
```

A STEP OUT command was issued when the program's main procedure was the only active procedure.

174

```
Source assign name must include a volume name
```

The original and new names you use in an ADD SOURCE ASSIGN command must include a volume name.

176

```
Source assign original name and current name must have the
same qualification
```

The original name and new name you use in an ADD SOURCE ASSIGN command must both be qualified to the same level. If the original name is a volume, the new name must be a volume; if the original name is a subvolume, the new name must be a subvolume.

177

```
This source assign name does not exist
```

The original name you provided in a DELETE SOURCE ASSIGN command does not have a source assignment. Use the LIST SOURCE ASSIGN command to display the existing source assignments.

183

```
Invalid length specified for DISPLAY
```

The value in a FOR clause of a DISPLAY command evaluated to a negative number. This number must be a non-negative value.

184

```
Data page containing variable was not saved
```

This error may be reported when examining save files created from system dumps. It is reported when the specified variable is contained in a data page that was not present at the time that the system dump was taken. (This only applies to save files that were created from a system dump by CRUNCH.)

185

```
Unable to trace subproc(s) due to 'S' register modification
```

Inspect cannot display all subprocedures entries in the call history because the program has an unexpected S register value. The stack markers that link subprocedure calls are addresses relative to the S register. If the program modifies the S register, Inspect cannot locate subprocedures stack markers.

188

SSGET: *error-specification*

The SSGET system procedure returned the reported error number. See the *Guardian Procedure Errors and Messages Manual* for interpretation of the error number.

189

```
SPI buffer too large
```

The DISPLAY command specified the SPI-NUM or EMS-NUM clause to display an SPI buffer whose used length exceeded 4K bytes.

190

SSID conversion error *n-n*

An error occurred on a call to the TEXTTOSSID or SSIDTOTEXT procedure. The n-n values are the error status codes returned by the procedure. For a description of these codes, see the *Guardian Procedure Calls Reference Manual*.

191

```
Current location is not a subproc
```

A STEP OUT SUBPROCS command was issued, but the current location is not within a subprocedure.

192

```
Unable to step subproc(s) due to 'S' register modification
```

A STEP OUT command was issued from a subprocedure, but Inspect cannot step out of the subprocedure(s) because the program has modified the S register. The stack markers that link subprocedure calls are addressed relative to the S register. If the program modifies the S register, Inspect cannot locate the subprocedure stack marker.

To step out of a subprocedure, set a temporary breakpoint on the statement following the subprocedure call and issue a RESUME command. To step out of a containing procedure, issue a STEP OUT PROC command.

194

```
Target must be a permanent disk file: file-name
```

An ADD SOURCE ASSIGN command was issued with a process or file for the original name, but the new name was not a permanent disk file. The new name must be a permanent disk file because the SOURCE can only read from permanent disk files.

195

```
Invalid operation in deleted program
```

An operation was issued for a deleted program. When you enter a STOP command, Inspect awaits confirmation of the stop from the operating system before removing the program from the program list. Therefore, the stopped program might still appear on the program list for a brief period of time.

196

```
Invalid timestamp
```

Inspect encountered an invalid timestamp (the year was greater than 4000).

197

```
Location deleted due to optimizations
```

Inspect encountered a statement that was deleted during optimization. If optimization across statements (using compiler directive OPTIMIZE 2) was performed, usually some statements are deleted. Because instructions for deleted statements do not exist, certain Inspect commands cannot be applied to them.

For accelerated programs executed on a TNS/R machine, Inspect only allows TNS code breakpoints to be set at locations that are memory-exact points. If you attempt to set a TNS code breakpoint at a location that is not a memory-exact point you will receive this error.

198

```
Results might be unexpected due to optimization.
```

If optimization across statements was performed, the instructions for statements are sometimes merged. Therefore, some Inspect operations (for example, MODIFY) might not produce the expected results.

199

```
System is unavailable
```

The system is unavailable because all network paths to it are down.

201

```
No programs exist
```

The LIST PROGRAM command found no programs on the program list for the current session.

202

```
No breakpoints exist
```

The LIST BREAKPOINT command found no breakpoints defined in the current program.

203

```
String not found
```

The search text specified in the FC, XC, or LIST HISTORY command was not found in the history buffer.

204

```
Command number command-number has not yet been entered
```

The FC, XC, or LIST HISTORY command specified a command number that does not yet exist.

205

```
History buffer is empty
```

The FC, XC, or LIST HISTORY command found the history buffer empty.

206

```
Command number not found
```

The FC, XC, or LIST HISTORY command specified a command number that is no longer in the history buffer.

207

```
Call history ordinal exceeds the number of calls
```

The command list specified a scope number greater than the number of calls in the call history. The TRACE command lists legal scope ordinals.

208

```
Invalid function key specified
```

The ADD KEY, LIST KEY, DELETE KEY, or FK command specified an invalid function key.

209

```
Unknown function key
```

You pressed a function key that Inspect does not support.

210

```
Function key undefined
```

The function key you pressed has no definition.

211

```
Function key buffer full
```

The ADD KEY command specified too long a replacement string.

212

```
Breakpoint not found
```

The CLEAR, FB (fix breakpoint), or LIST BREAKPOINT command specified a breakpoint that is not defined in the current program.

213

```
Alias not found
```

The LIST ALIAS or DELETE ALIAS command specified an alias name that is not defined in the current session.

214

```
Alias table empty
```

The LIST ALIAS or DELETE ALIAS command found no aliases defined in the current session.

215

```
Invalid breakpoint number
```

The command list specified a breakpoint number less than 1 or greater than 99.

216

```
Function key buffer empty
```

The LIST KEY command found no function keys defined in the current session.

218

```
Command invalid in profile
```

The INSPLOCL file or the INSPCSTM file included one of these commands, which are not valid in customization files:

ADD PROGRAM
EXIT
HOLD
PAUSE
RESUME
SELECT PROGRAM

STEP
STOP

219

```
Command reserved for interactive users
```

The command list, which Inspect read from an OBEY file, INSPLOCL, or INSPCSTM, included a command that can only be used interactively.

220

```
Type too large for register
```

The DISPLAY REGISTER command specified a display type that requires more data than a register contains.

221

```
Previous assign for original name replaced
```

The ADD SOURCE ASSIGN command specified an original name that already had an assignment. Inspect replaced the assignment.

222

```
SOURCE ASSIGN list is empty
```

The LIST SOURCE ASSIGN command found no source assignments defined in the current session.

223

```
SOURCE OPEN list is empty
```

The LIST SOURCE OPEN command found no open source files.

224

```
Type too large for value
```

The DISPLAY VALUE command specified a display type that requires more data than specified in the VALUE clause.

228

```
Ordinal statement-number is less than 1, not a valid ordinal
```

The command list specified a statement number less than one.

229

```
Ordinal statement-number exceeds the number of statements in
the scope
```

The command list specified a statement number beyond the last statement in the scope unit.

231

```
Too many items in STATUS list
```

The SET STATUS LINE25 or SET STATUS SCROLL command specified too many status items.

232

```
program is not a process
```

The ADD PROGRAM command did not specify a process or a save file as the program to add.

233

```
Process not found
```

The ADD PROGRAM command specified a nonexistent process.

234

```
FILE and LOC clause not allowed with STATEMENT ordinals
```

The SOURCE STATEMENT command specified a FILE or LOC clause. These clauses are not valid in the SOURCE STATEMENT command.

235

```
File position range (F or L) not allowed
```

The SOURCE ICODE command specified a range using the F (first line) or the L (last line) position option. These options cannot be used with SOURCE ICODE.

237

```
SPI FORMATTING ERROR: error
```

The SPI_BUFFER_FORMATSTART_, SPI_BUFFER_FORMATNEXT_, or SPI_BUFFER_FORMATFINISH_ system procedure returned the reported error number. For more information about the interpretation of the error number, see the *Guardian Procedure Errors and Messages Manual*.

238

```
Invalid data location- Use INFO LOCATION for code locations
```

A code location was specified with the INFO IDENTIFIER command. Use INFO LOCATION to retrieve information about code locations.

239

```
No segments are allocated
```

This warning is issued when the INFO SEGMENTS command did not find any extended segments allocated for the current program.

240

```
File is not open: file-number
```

A file number that is not open was specified with the INFO OPENS command.

241

```
No files are open
```

This warning is issued when the INFO OPENS command found no files opened by the current program.

253

```
Search pattern required
```

A search string was not specified with the SOURCE SEARCH command.

254

```
Search clause already specified
```

A parameter or clause was specified more than once with the SOURCE SEARCH command.

259

```
File already exists: file-name
```

This error is reported when the file specified with the SAVE command already exists. If you want to overwrite the file, specify a “!” following the file name.

273

```
Target field exceeds TCP modification limit
```

The amount of data specified when you modified a variable of a SCOBOL program exceeded the limit of the TCP.

274

```
Process has disabled STOP; it will be stopped after it
reenables STOP
```

This error is reported if you issue a STOP command and the process has called SETSTOP to disable external stop requests. This is usually done when a process is performing critical operations that should not be interrupted. You should resume the process to allow it to complete these operations. The process will stop when it calls SETSTOP again to enable external stop requests.

275

```
Command names may not be used as aliases
```

This error is reported if you attempt to define an alias having the same name as an Inspect command or abbreviation.

276

```
Expression items must be element level: expression
```

This error is reported when a structure or record variable is used in an expression and an element was not specified.

278

```
Current language might change at the next event
```

You will receive this warning if you use SET LANGUAGE instead of SELECT LANGUAGE.

279

```
system-name has been selected as the SOURCE SYSTEM for the
current program only
```

This warning is issued when the SET SOURCE SYSTEM command is issued rather than the SELECT SOURCE SYSTEM command. This SET command only applies to the current program; not to the Inspect session as SET commands usually do.

280

```
The SOURCE SYSTEM has been cleared for the current program
only
```

You typed SET SOURCE SYSTEM and carriage return.

292

```
Variable is local to inactive subprocedure: variable name
```

This error is issued if you specify a variable that is local to a TAL subprocedure that is currently not active. You can use the TRACE command to display the procedures and subprocedures that are currently active.

293

```
Unable to convert expression to a TRUE or FALSE value
```

The requested operation demanded the conversion of an expression to a true or false value and Inspect could not perform the conversion. For example, an expression used in the BREAK command's IF clause must result in a true or false value.

294

```
Unable to convert the expression to the type of the variable
being modified
```

The requested operation required a conversion between incompatible types.

295

```
A file number must be an integer expression
```

The requested operation required a conversion between incompatible types. For example, a file number must be a positive integer value and Inspect was unable to convert the entered expression into a integer value.

296

```
No address for variable ID
```

The specified variable is a type for which no address information is available. Inspect does not have access to the variable.

297

```
Step count maximum is 32767
```

The count specified in a STEP command must be a positive integer less that or equal to 32767.

298

```
No such instance of subprocedure on stack
```

You requested access to a particular activation of a subprocedure. This particular instance does not exist. For example, DISPLAY *subproc^ name(3).sublocal* would get this message when there are only two instances of subproc^name on the stack.

299

```
A subprocedure instance must be a signed 16-bit integer:
subprocedure name
```

The number specified for a subprocedure instance must be a signed 16-bit integer. For example, the instance shown here, DISPLAY *subproc^name* (3000000).*sublocal*, is invalid.

300

```
Unable to traverse subprocedure calls
```

Inspect was unable to find a value on the data stack that matched a subprocedure's return address. This could be due to a program’s manipulation of the “S” register within a subprocedure.

301

```
Segment number maximum is 32767
```

The number specified in the command exceeded the allowed range. Segment numbers must be positive, signed 16-bit numbers.

302

```
RP value must be in the range 0-7
```

The number specified in the command exceeded the allowed range. The register pointer value in the RP clause of the RESUME AT command must be in the range 0-7; numbers in that range are the only legal register values.

303

```
A CPU value must be in the range 0-15
```

The maximum number of CPUs in a Tandem system is 16, and they are numbered from 0-15. The number you specified in the command exceeded the allowed range.

304

```
A PIN value must be in the range 0-255
```

The maximum number of processes in a CPU is 256. They are numbered from 0 to 255. The number you specified in the command exceeded the allowed range.

305

```
Expression is not scaled to match pointer type
```

This is a warning. Inspect cannot scale a constant to match the pointer type in an expression.

306

```
[ACCESS/CHANGE/WRITE] is not allowed with a code location
```

You attempted to set a breakpoint on a code location when the break event specified is not allowed, that is, ACCESS, CHANGE, or WRITE on a code breakpoint.

307

```
Internal Frame Error
```

DMON and Inspect encountered an error attempting to retrieve stack frames. You cannot access data from frames other than the current location frame. Use the SELECT DEBUGGER DEBUG command followed by TN as an alternative.

350

```
Breakpoint was not set in this INSPECT session.
```

An FB (fix breakpoint) command was issued on a breakpoint that was set from Debug or a previous Inspect session.

351

```
TNS program location is unavailable
```

When you are debugging an accelerated program on a TNS/R system, this error is issued if the specified code location does not have a corresponding TNS/R code location.

352

```
Invalid operation on a TNS system
```

This error is issued if you attempt to perform an operation that is specific to a TNS/R system when debugging a program executing on a TNS system.

353

```
Current location is accelerated; stepping instructions is not
allowed.
```

When debugging an accelerated program on a TNS/R or TNS/E system, this error is issued if you attempt to step execution in terms of instructions. Accelerated programs can only be stepped using source-level units.

354

```
The values in registers R0-R7 and E are out of date
```

This warning is reported for accelerated programs when TNS registers are displayed and the current location is not a register-exact point. TNS register values are only guaranteed to be accurate at register-exact points.

355

```
Current location is not a register-exact point; registers
cannot be modified
```

This error is reported for accelerated programs when a command is issued to modify a TNS machine register and the current location is not a register-exact point.

The usefulness of modifying TNS registers when debugging accelerated programs is significantly limited by the fact that they can only be modified at register-exact points.

356

```
Value must be the address of a register-exact point
```

This error is reported for accelerated programs when you attempt to modify the TNS P register and specify an address value that is not the address of a register-exact point.

You can use the high-level ICODE and the low-level I command to determine if there are any register-exact points near the address; instructions at register-exact points are marked with a “@.”

357

```
Target location must be a register-exact point
```

This error is reported for accelerated programs if the target location specified with a RESUME AT command is not a register-exact point.

358

```
Modify may have no effect; data that is about to be used may
be stored in registers
```

This warning is reported for accelerated programs when you issue a command that modifies memory and the current program location is not a register-exact point. Under these circumstances, data that is about to be used may have already been loaded from memory into registers, in which case modifying the value in memory would have no effect on program behavior.

Inspect cannot predict when a modify operation may have no effect. Chances are highest when you are modifying a variable that has been used recently or is about to be used.

359

```
Current location is not a memory-exact point;
displayed values may be out of date;
the location reported is an approximate TNS location
```

This warning is reported for accelerated programs when Inspect reports a debugging event and the current location is not a memory-exact point. In this case, some values in memory may be out of date, because more recent values are stored in registers. This is most likely to be the case for variables to which values have been recently stored.

Most commonly, programs are left at non-exact points as a result of data access breakpoints. Issuing a STEP command will advance the program to the next memory-exact point, at which displayed memory will be consistent.

360

```
DMON internal error: number
Contact your Tandem representative
```

The DMON process reported an internal error. You should record any information that is reported and contact your Tandem representative. Such an error will prevent you from performing the operation that you attempted; however, other operations should still work.

361

```
Operation available only on an accelerated program
```

This error is reported if you attempt an operation that only applies to an accelerated program when you are debugging a program that has not been accelerated or is not executing on a TNS/R system.

362

```
Object file was created by an unrecognized Accelerator
version; file: file-name
```

The version of the accelerator used to create the object file is a version that is not recognized by Inspect. You may need to use a newer version of Inspect.

363

```
Variable is larger than 16-bits in size
breakpoint is set on the first 16-bit word
```

This warning is reported if you set a data breakpoint on a variable that is larger than 16 bits in size. In this case, the breakpoint can only be set on the first word of memory that the variable occupies.

364

```
Value of variable did not change; breakpoint may have been
triggered by an access to the containing 16-bit word
```

This warning is only reported for write memory access breakpoints. It is reported when the value of the variable has not changed and the size of the variable is less than 16 bits. In this case, the breakpoint may have been triggered by a store to the portion of the word that does not contain the variable. Inspect cannot determine if the breakpoint was triggered by a store or if you simply stored the same value in the variable.

365

```
Machine does not support write memory access breakpoints; use
change breakpoints
```

This error is reported if you attempt to set a write memory access breakpoint on a NSR-L processor. You must set either a change breakpoint or a read/write breakpoint. Refer to the discussion of data breakpoints in Section 15, Using Inspect on a TNS/R System, for important considerations in using data breakpoints on TNS/R systems.

366

```
Information about all open files could not be saved
```

This warning is reported when loading a save file in which it was not possible to save information about all the open files that a process had open at the same time. This only occurs when a process has had more than 500 files open.

367

```
All data segments could not be saved
```

This warning is reported when loading a save file in which it was not possible to save the information about all the extended data segments that a process had allocated. This occurs when the amount of extended data allocated by the process is greater than the capacity of a file. Segments are saved in the order that they were allocated, so, in this case, data for segments allocated later may not be available.

368

```
The current program is not a save file
```

The command that was issued can only be used when the current program is a save file.

369

```
The RP clause cannot be used with an accelerated program
```

This error is reported for accelerated programs if you attempt to use the RP clause with the RESUME AT command. This clause is not supported for accelerated programs since the TNS register pointer is only updated and used at certain locations.

370

```
Current location must be a register-exact point
```

This error is reported for accelerated programs when the current location must be a register-exact point. For instance, this requirement exists when using the RESUME AT command to cause the program to begin execution at a specified location.

371

```
Program is not executing TNS/R instructions
```

This error is reported for accelerated programs by commands that require the program to be executing TNS/R instructions. For instance, you can only display or modify TNS/R registers in an accelerated program when the program is executing TNS/R instructions.

The LIST PROGRAM DETAIL command displays whether the program is currently executing TNS/R or TNS instructions.

372

```
Starting location is not a memory-exact point;
listing begins at preceding memory-exact point
```

This warning is reported for accelerated programs by the ICODE command when it is necessary for a TNS address to be mapped to a TNS/R address and the specified location is not a memory-exact point. Most commonly, this occurs when BOTH is specified to list the correspondence between TNS and TNS/R instructions. It can also occur when listing TNS instructions given a TNS/R address or listing TNS/R instructions given a TNS address.

374

```
A memory access breakpoint has left the program at a location
where execution cannot be stepped
```

This error is reported only for accelerated programs in the rare instance when a memory access breakpoint, or data breakpoint, has suspended the program in millicode routine that will alter the flow of program execution. If this is reported, set a temporary breakpoint at the appropriate location and resume program execution.

375

```
Memory access breakpoints not allowed with FB or AS COMMANDS
```

You cannot use the FB command or AS COMMANDS clause with data (memory access) breakpoints.

376

```
CHANGE not allowed with READ or WRITE
```

You cannot specify READ or WRITE clauses in combination with the CHANGE clause when setting a data breakpoint.

377

```
Cannot select program until it becomes the primary
```

This error is reported if you attempt to select a program that has been added to the program list as a result of setting a breakpoint in the backup process of a fault-tolerant process pair. You must wait until the program becomes the primary process.

378

```
Insert or replace would have caused command line to exceed
255 characters
```

This warning is reported by the FA, FB, FC and FK commands when text insertion or replacement would cause the command line length to exceed 255 characters.

379

```
The following was formatted for display when an error was
encountered
```

This warning is reported by the DISPLAY command if an error occurred while formatting data to be displayed. In such a case, any values that were formatted before the error occurred are displayed.

380

```
Recursive alias definition
```

This warning is reported if you attempt to define an alias that calls itself.

381

```
No extended data segment currently selected
```

This warning is reported by the DISPLAY command if the specified variable or address resides in an extended data segment and there is no extended data segment currently selected.

You can use the INFO SEGMENTS command to list the extended data segments currently allocated by the program and the SELECT SEGMENT command to select the extended segment that Inspect is to use.

382

```
The saved environment register settings are out of date
```

This warning is reported for accelerated programs by the TRACE REGISTERS command. For accelerated programs, saved TNS environment register values may be out of date.

383

```
Value of variable did not change
```

This warning is reported when a write data breakpoint is triggered and the value of the variable on which the breakpoint is set did not change. In this case, the breakpoint might have been triggered by the program storing the same value into the variable.

384

```
Low-level breakpoints not allowed with FB or AS COMMANDS
```

This error is reported if an attempt is made to use the FB command or the AS COMMANDS clause with breakpoints that were set from low-level Inspect.

385

```
Object file file does not contain accelerator information
```

This warning is reported by the ADD PROGRAM command when a save file for an accelerated program is added and the file specified with the CODE clause does not contain accelerator information. When this is the case, you can examine most state information contained in the save file, with the exception of information that is specified to accelerator programs.

386

```
Backup for program has taken over
```

This warning is issued when Inspect reports an event and the backup process of a fault-tolerant process pair in which a backup breakpoint is set has taken over.

387

```
Ending location is not a memory-exact point;
listing ends at following memory-exact point
```

This warning is reported for accelerated programs by the ICODE command when it is necessary for a TNS address to be mapped to a TNS/R address and the end of the specified range is not a memory-exact point. Most commonly, this occurs when BOTH is specified to list the correspondence between TNS and TNS/R instructions. It can also occur when listing TNS instructions given a TNS/R address or listing TNS/R instructions given a TNS address.

388

```
Listing ends at procedure end
```

This warning is reported by the ICODE command when a source-level location was specified and the specified range exceeded the bounds of the procedure. In this case, ICODE is listed until the end of the procedure is reached.

389

```
Address is not aligned to a TNS/R instruction boundary
it will be rounded down to the previous instruction
```

This warning is reported for accelerated programs if the specified TNS/R address is not 32-bit word aligned.

390

```
Variable is larger than 16-bits and is not word aligned;
breakpoint is set on the first 16-bit word containing the
variable
```

This warning is reported when a memory access breakpoint is set on a variable that is that is not word aligned larger than 16 bits in size. In this case, the breakpoint is set on the first word of memory that contains the variable.

391

```
Current location is not a memory-exact point;
execution will be stepped to the next memory-exact point
```

This warning is reported for accelerated programs when a STEP command is issued and the current location is not a memory-exact point. Under these circumstances, the STEP command will advance the program to the next memory-exact point.

392

```
Object file file was created by an unrecognized Accelerator
version;
Accelerator information will be unavailable
```

This warning is reported when a save file for an accelerated program is added and the object file was created by a version of the accelerator that is not recognized by Inspect. You will still be able to access program state information except that which is specific to accelerated programs.

393

```
A TNS/R register must be in the range 0-31
```

This error is reported if you attempt to specify a TNS/R machine register that is not in the range \$0 - \$31.

394

```
Non-existent program object file: file
```

This error is reported when using the ADD PROGRAM command to add a save file and the program file specified in the save file cannot be found. You must use the CODE clause of the ADD PROGRAM command to specify where the object file can be found.

395

```
Non-existent library object file: file
```

This error is reported when using the ADD PROGRAM command to add a save file and the user library file specified in the save file cannot be found. You must use the LIB clause of the ADD PROGRAM command to specify where the object file can be found.

396

```
Safeguard error [nnn on $ZSMP], save file might retain
SUPER.SUPER default protection
```

This warning is reported when a error nnn occurs during communication to the SAFEGUARD SPI process ($ZSMP), or when an SPI error is returned from $ZSMP.

397

```
Optimization level for scope <name> is not supported by
Inspect
```

Inspect supports TNS/R native compiler optimizations 0 and 1. When debugging TNS/R native programs compiled with optimization level 2, output and actions may be unexpected for commands DISPLAY, MODIFY and STEP.

406

```
Process uses dynamic link libraries.  Visual Inspect is
required.
```

This error message is reported when the user enters an ADD PROGRAM command and attempts to add a process that uses DLLs. Inspect does not support the debugging of DLLs. You must use either Visual Inspect or Debug on TNS/R systems. On TNS/E systems, use either Visual Inspect or Native Inspect.

450

```
This snapshot file contains dynamic link library information.
Visual Inspect is required.
```

This error message is reported when the user enters an ADD PROGRAM command and attempts to open a snapshot file that contains DLL information. Inspect does not support the debugging of DLLs. You must use either Visual Inspect or Debug on TNS/R systems. On TNS/E systems, use either Visual Inspect or Native Inspect.

1003

```
Process name cpu,pin is a TNS/E process.  Visual Inspect or
Native Inspect is required.
```

This error is reported when the ADD PROGRAM command is used to add a TNS/E process to an Inspect debugging session on a TNS/E system. Inspect can only be used to debug TNS programs and TNS and TNS/R snapshots on a TNS/E system.

1004

```
This is a snapshot of a TNS/E process.   Visual Inspect or
Native Inspect is required.
```

This error is reported when the ADD PROGRAM command is used to add a TNS/E snapshot to a debugging session on a TNS/E system. Inspect can only be used to debug TNS programs and TNS and TNS/R snapshots on a TNS/E system.

1005

```
Inspect cannot display TNS/E registers.
```

A command such as the DISPLAY REGISTER BOTH command was entered, and the current program is a TNS program on a TNS/E system. Inspect will display the TNS registers as if the DISPLAY REGISTER ALL command was entered.

1006

```
Inspect cannot display TNS/E instructions.
```

A command such as the ICODE *`address`* BOTH command was entered, and the current program is a TNS program on a TNS/E system. Inspect will display the TNS instructions as if the command `ICODE` *`address`* was entered, where *`address`* is a TNS address.

1007

```
Inspect cannot read a TNS/E object file.
```

A command such as INFO SAVEFILE was entered, specifying a TNS/E object file. To debug TNS/E native object files, use either Visual Inspect or Native Inspect.

1008

```
BACKUP option is not supported on TNS/E.
```

The BREAK command was specified with the BACKUP option to debug a backup process. On TNS/E, the backup process is not owned by the debugger when the user tries to set the backup breakpoint.

1009

```
STEP OUT count is ignored when process is at a TNS/E address.
```

A TNS process can be suspended at a TNS/E location if, for example, the process hits a data breakpoint in TNS/E system library code. If you enter STEP OUT *number*, the number is ignored and Inspect returns the process to the TNS call site.

1106

```
Incompatible snapshot file version. Native Inspect is
required.
```

This error occurs when the ADD PROGRAM command is used to add a TNS/E 64-bit snapshot to a debugging session in a TNS/E system. Inspect is used only to debug TNS programs and both TNS and TNS/R snapshots on a TNS/E system.

1107

```
Process name cpu,pin is a TNS/E 64-bit process. Native
Inspect is required.
```

This error occurs when the ADD PROGRAM command is used to add a TNS/E 64-bit process to an Inspect debugging session in a TNS/E system. Inspect is used only to debug TNS programs and both TNS and TNS/R snapshots on a TNS/E system.

B Syntax Summary

High Level Inspect Commands

The syntax of the four following high-level parameters depends on the programming language you are using:

```
scope-path code-location data-location expression
```

The language subsections of this appendix describe these parameters.

ADD

```
ADD list-item

list-item: one of

   ALIAS alias-name [=] command-string
   KEY key-name [=] command-string
   PROGRAM program-spec
   SOURCE ASSIGN [ original-name ,] new-name

   command-string:  one of

      " [ character ]... "
      ' [ character ]... '

   key-name:  one of

      F1   F2   F3   F4    F5   F6    F7   F8
      F9   F10  F11  F12   F13  F14   F15  F16
      SF1  SF2  SF3  SF4   SF5  SF6   SF7  SF8
      SF10 SF11  SF12 SF13   SF14  SF15  SF16
   program-spec:  one of

      process

      save-file [ CODE code-file ]
                [ LIB lib-file ]
                 [ SRL {( srl-file [ , srl-file,...])}]

   original-name:  one of

      [ \system. ] $volume [ .subvolume [ .file ] ]
      [ \system. ] $process [ .#qual-1 [ .qual-2 ] ]
      [ \system. ] cpu, pin
      [ \system. ] $volume.#number
      /oss-pathname [/oss-pathname...]

   new-name:  either of

      [\system.] $volume [ .subvolume [ .file ] ]

      /oss-pathname [/oss-pathname...]
```

ALIAS

```
ALIAS[ES] [ alias-name [ , [ = ] replacement string ] ]

replacement-string:  one of

      " [ character ]..." ]
      ' [ character ]...' ]
```

BREAK

```
BREAK [ breakpoint [ , breakpoint ]...]

breakpoint:

      brk-location [ brk-condition ]... [ brk-action ]...

   brk-location:

      { code-location } [ BACKUP ]
      { data-location }
      { [#] ABEND }
      { [#] STOP }

   brk-condition:  one of

      EVERY integer
      IF expression
```

data-subtype: one of

```
ACCESS   CHANGE   READ   WRITE   READ WRITE   WRITE READ
```

brk-action: one of

```
TEMP [ integer ]

THEN { command-string | alias-name }
```

CD

```
CD { oss-pathname }
```

CLEAR

```
CLEAR { * | clear-spec }
```

clear-spec: one of

```
breakpoint-number [ , breakpoint-number ]
CODE code-location [ , code-location ]
DATA data-location [ , data-location ]
EVENT { ABEND | STOP }
```

COMMENT

```
COMMENT | -- [ text ]
```

DELETE

```
DELETE list-item
```

list-item: one of

```
ALIAS[ES] { * | alias-name }
KEY[S] { * | key-name }
SOURCE ASSIGN { * | original-name }
SOURCE OPEN[S] { * | source-file }
```

key-name: one of

```
F1    F2   F3   F4    F5   F6    F7   F8
F9    F10  F11  F12   F13  F14   F15  F16
SF1   SF2  SF3  SF4   SF5  SF6   SF7  SF8
SF9   SF10  SF11  SF12  SF13  SF14  SF15  SF16
```

original-name: one of

```
[ \system. ] $volume [ .subvolume [ .file ] ]
[ \system. ] $process [ .#qual-1 [ .qual-2 ] ]
[ \system. ] cpu, pin
[ \system. ] $volume.#number
/oss-pathname [/oss-pathname...]
```

DISPLAY

```
DISPLAY item [ , item ]... [ formatting-clause ]
```

item: one of

```
display-data [ WHOLE ] [ PLAIN ] [ FOR for-spec ]
display-code [ FOR for-spec ]
REGISTER register-item [ TYPE display-type ]
spi-buffer
spi-token [ AS data-type ] [ FOR for-count ]
string
( expression )
VALUE value-list [ TYPE display-type ]
```

display-data: one of

```
identifier
data-location AS data-type
data-location TYPE display-type
```

identifier

specifies constants and variables defined by the program.

data-location: one of

```
( expression ) [ SG ]
identifier
```

display-type: one of

```
CHAR             CRTPID          DEVICE        ENV
FILENAME        FILENAME32      FIXED         FLOAT
INT             INT16           INT32         LOCATION
PROCESS HANDLE  REAL            REAL32        REAL64
SSID            STRING          SYSTEM        TIMESTAMP
TIMESTAMP48     TOSVERSION      TRANSID       USERID
USERNAME
```

for-spec:

```
for-count [ BYTE[S] | WORD[S] | DOUBLE[S] | QUAD[S] ]
```

display-code: one of

scope-path

```
[ scope-path ] code-reference

code-reference:  one of
   scope-unit
   label
   #line-number

for-count: one of
   non-negative integer
   data-location

register-item:  one of
   ALL       BOTH      TNS       TNS/R      register-name

register-name:  one of
   tns-register-name
   tns/r-register

   tns-register-name:  one of
      P     E     L     S
      R0    R1    R2    R3    R4    R5    R6    R7
      RA    RB    RC    RD    RE    RF    RG    RH

   tns/r-register:  one of
      $PC    $H1   $LO    $0    $1...$31
      tns/r-register-alias

      tns/r-register-alias:  one of
         $AT   $V0   $V1   $A0   $A1   $A2  $A3
         $S0   $S1   $S2   $S3   $S4   $S5  $S6 $S7
         $T0   $T1   $T2   $T3   $T4   $T5  $T6 $T7 $T8 $T9
         $K0   $K1   $GP   $SP   $FP   $RA

spi-buffer:
   data-address TYPE spi-type
```

```
    spi-type: one of
       EMS   EMS-NUM   SPI   SPI-NUM

 spi-token:
    data-address : token-spec [ TYPE spi-type ]
    [ POSITION token-spec [ , token-spec ]... ]

    token-spec:
       token-code [ : token-index ] [ SSID ssid-string ]

       token-code: one of
          token-index
          ssid-string

 value-list: one of
    integer
    integer , integer
    integer , integer , integer , integer

 formatting-clause: one of
    IN base [ base ]...
    { FORMAT | FMT } format-list
    PIC mask-string [ , mask-string ]...

 base: one of
    BINARY  OCT[AL]  DEC[IMAL]  HEX[ADECIMAL]
    ASCII   XASCII   GRAPHICS
    ICODE

 format-list:
    an edit-descriptor list for the operating system formatter

 mask-string:
    a mask string for the M edit descriptor
```

ENV

```
ENV [ env-parameter ]
```

env-parameter: one of

```
DIRECTORY
LANGUAGE
LOG
PROGRAM
SCOPE
SOURCE SYSTEM
SYSTEM
SYSTYPE
VOLUME
```

EXIT

```
EXIT
```

FA

```
FA alias-name
```

FB

```
FB breakpoint-number
```

FC

```
FC [ command-line-specifier ]
```

command-line-specifier: one of

```
pos-num
neg-num
search-text
" search-text"
```

FILES

```
FILES [ { * | file-list > } [ DETAIL ] [ file-type ]
```

```
file-list:
   file-number [, file-number ]

file-type:
   FORTRAN   FD  GUARDIAN
```

FK

```
FK key-name
   key-name:  one of
      F1 F2 F3 F4 F5 F6 F7 F8
      F9 F10 F11 F12 F13 F14 F15 F16
      SF1 SF2 SF3 SF4 SF5 SF6 SF7 SF8
      SF9 SF10 SF11 SF12 SF13 SF14 SF15 SF16
```

HELP

```
   HELP [ topic ]
   topic:
      main-topic [ sub-topic [ sub-topic ] ]
```

HISTORY

```
   HISTORY [ num ]
   num
```

HOLD

```
   HOLD [ program [ , program ]... ]
        [ * ]
   program:  one of
      program-number
      program-name
      cpu, pin
```

ICODE

```
ICODE location [ FOR count [ unit ] ] [ report ]
location:  one of
   [ AT code-location ]
   [ tns/r tns/r-address-expression ]
   [ tns-address-expression ] [ UC.number | UL.number ]
unit:  one of
   INSTRUCTION[S]
   STATEMENT[S]
   VERB[S]
report:  one of
   TNS
   tns/r
   BOTH
tns/r:  one of
   TNSR
   TNS/R
   R
tns/r-address-expression:
   tns/r-value [ operator tns/r-value ]...
tns-address-expression:
   tns-value [ operator tns-value ]...
operator:  one of
   * / << >> + -
tns-value:  one of
   ( tns-expression )
   16-bit number
   tns-register
tns/r-value:  one of
   ( tns/r-address-expression )
   32-bit number
   16-bit number [ .16-bit number ]
   tns/r-register
```

IDENTIFIER

```
IDENTIFIER { * | identifier-spec }
identifier-spec: one of
   [ scope-path. ] identifier [ .identifier ]...
   scope-path
   #data-block
   ##GLOBAL
```

IF

```
IF expression THEN { command | alias-name }
```

INFO

```
INFO info-item
info-item: one of
   IDENTIFIER { * | identifier-spec }
   LOCATION [ * | [ SCOPE scope-path | scope-ordinal ] ]
   OBJECTFILE [ FILE filename ]
   OPENS [ { * | file-list } [ DETAIL ] [ file-type ]
   SAVEFILE [ FILE filename ]
   SCOPE [ scope-number | scope-path ]
   SEGMENT[S] [ * | segment-id ] [[, ] DETAIL ]
   SIGNALS[S] [ * | signal-id [, signal-id ...] ]
identifier-spec: one of
   scope-path
   [ scope-path. ] identifier [ .identifier ]...
   #data-block
   ##GLOBAL
file-list:
   file-number [, file-number ]
```

KEY

```
KEY[S] [ key-name [ [=] replacement-string ] ]
key-name: one of
   F1 F2 F3 F4 F5 F6 F7 F8
   F9 F10 F11 F12 F13 F14 F15 F16
   SF1 SF2 SF3 SF4 SF5 SF6 SF7 SF8
   SF9 SF10 SF11 SF12 SF13 SF14 SF15 SF16
```

LIST

```
LIST list-spec [ AS COMMAND[S] ]
list-spec: one of
   ALIAS[ES] [ alias-name ]
   BREAKPOINT[S] [ breakpoint-number ]
   HISTORY [ command-range ]
   KEY[S] [ key-name ]
   PROGRAM[S] [ program ]
   SOURCE ASSIGN[S]
   SOURCE OPEN[S]
options: one of
   [ [ , ] DETAIL ] | [ AS COMMAND[S] ]
key-name: one of
   F1 F2 F3 F4 F5 F6 F7 F8
   F9 F10 F11 F12 F13 F14 F15 F16
   SF1 SF2 SF3 SF4 SF5 SF6 SF7 SF8
   SF9 SF10 SF11 SF12 SF13 SF14 SF15 SF16
program: one of
   program-number
   program-name
```

LOG

```
LOG { [ BOTH | INPUT | OUTPUT | AS COMMAND[S]] TO file-name }
    { STOP }
```

LOW

```
LOW
```

MATCH

MATCH { SCOPE *pattern* }

    { IDENTIFIER *pattern* [[,] SCOPE *scope-spec* |[,VERBOSE]]}

scope-spec: one of

  scope-number

  scope-path

MODIFY

```
MODIFY { data-location [ WHOLE ] [ { = | := } mod-list ] }
       { REGISTER register-name [ { = | := } expression ] }
       { SIGNAL signal-id [ { = | := } signal-list ] }
mod-list:
   mod-item [ , mod-item ]...
mod-item:
   [ integer COPIES ] expression
register-name:  one of
   tns-register-name
   tns/r-register
tns-register-name:  one of
   P E L S
   R0 R1 R2 R3 R4 R5 R6 R7
   RA RB RC RD RE RF RG RH
tns/r-register:  one of
   $PC $H1 $LO $0 $1...$31
signal-id:  one of
   SIGABRT SIGALRM SIGFPE SIGHUP SIGILL SIGINT
   SIGKILL SIGPIPE SIGQUIT SIGSEGV SIGTERM SIGUSR1
   SIGUSR2 SIGCHLD SIGCONT SIGSTOP SIGTSTP SIGTTIN
   SIGTTOU SIGABEND SIGLIMIT SIGSTK SIGMEMMGR SIGNOMEM
   SIGMEMERR SIGTIMEOUT
signal-list:
   signal-handler, mask, flags
signal-handler:
   SIG_DEL | SIG_IGN | SIG_DEBUG | # function-name
mask:
   double [ double [ double [ double ]]]
flags:
   double
```

OBEY

```
OBEY file-name
```

OBJECT

```
OBJECT
```

OPENS

```
OPENS [ { * | file-list } [ DETAIL ] [ file-type ] ]
file-list:
   file-number [, file-number ]
file-type: one of
   FORTRAN FD GUARDIAN
```

OUT

```
{ OUT file-name }
{ /OUT file-name/ }
```

PAUSE

PAUSE

PROGRAM

```
PROGRAM[S] [ program [ CODE file-name ]
           [ LIB lib-file ]
           [ SRL {( srl-file [ , srl-file,...])}]
program: one of
   program-number
   program-name
   cpu, pin
```

RESUME

```
RESUME [ * [ EXIT ] ]
       [ program [ AT code-location [ , RP integer ] ] ]
program: one of
   program-number
   program-name
   cpu, pin
   Pathway-terminal-name
```

SAVE

```
SAVE filename [ ! ]
```

SCOPE

```
SCOPE [ scope-spec ]
scope-spec: one of
   scope-number
   scope-path [ ( instance) ]
   #data-block
   ##GLOBAL
```

SELECT

```
SELECT select-option

select-option:  one of

   DEBUGGER DEBUG

   SEGMENT [ segment id ]

   LANGUAGE language

   PROGRAM program [ CODE file-name ]
                   [ LIB file-name ]
                   [ SRL {( srl-file [ , srl-file,...])}]

   SOURCE SYSTEM [ \system ]

   SYSTYPE { GUARDIAN | OSS }

language:  one of

   C C++ COBOL COBOL85

   FORTRAN Pascal pTAL SCOBOL TAL

program:  one of

   program-number

   program-name

   cpu, pin
```

SET

```
SET set-assignment
set-assignment: one of
   CHARACTER FORMAT [=] { ASCII | GRAPHICS | XASCII }
   DEREFERENCE DEPTH [=] integer
   ECHO echo-item [=] { ON | OFF }
   LOCATION FORMAT [ level ] [=] loc-fmt [, loc-fmt ]...
   PRIV MODE [=] { ON | OFF }
   PROMPT [=] [ prompt-item [ , prompt-item ]... ]
   RADIX [ INPUT | OUTPUT ] [ level ] [=] radix
   SOURCE BACK [=] count
   SOURCE FOR [=] count
   SOURCE RANGE [=] range / range
   SOURCE WRAP [=] { ON | OFF }
   STATUS ACTION [ level ] [=] [ cmd-string ]
   STATUS LINE25 [ level ] [=] [ status-item-list ]
   STATUS SCROLL [ level ] [=] [ status-item-list ]
   SUBPROC SCOPING [=] [ SUBLOCAL | LOCAL ]
   SYSTYPE { GUARDIAN | OSS }
   TRACE trace-level [=] { ON | OFF }
echo-item: one of
   ALIAS[ES] HISTORY KEY[S]
level: one of
   BOTH HIGH LOW
loc-fmt: one of
   INSTRUCTION[S]
   LINE[S] [ FILE [ ALL ] ] [ OFFSET ]
   STATEMENT[S] [ OFFSET ]
prompt-item: one of
   string ACCELERATOR STATE COMMAND
   DIRECTORY FN ICODE
   INSTRUCTION SET LEVEL PROCESSOR
   PROGRAM FILE PROGRAM NAME PROGRAM ORDINAL
   PROGRAM PID RADIX SOURCE
   STEP SUBVOL[UME] SYSTEM
   SYSTYPE VOLUME
```
radix: one of
```
   DEC[IMAL] HEX[ADECIMAL] OCT[AL]
```
count:

   integer [STATEMENT[S] | LINE[S] | INSTRUCTION[S]]

range: one of

   F L #*line-number statement-number*

status-item-list:

   status-item [, *status-item*]...

status-item: one of

   string ACCELERATOR STATE EVENT
```
   INSTRUCTION SET LANGUAGE LOCATION
   NEW LINE PROCESSOR PROGRAM FILE
   PROGRAM NAME PROGRAM ORDINAL PROGRAM PID
   SCOPE STATE SYSTYPE
   TYPE
```
trace-level: one of
```
   ARGUMENT[S] SCOPE[S] STATEMENT[S]
```

SHOW

```
SHOW { ALL [ AS COMMAND[S] ] }
     { set-object }
set-object: one of
   CHARACTER FORMAT [=] { ASCII | GRAPHICS | XASCII }
   DEREFERENCE DEPTH integer
   ECHO { ALL | ALIAS[ES] | HISTORY | KEYS }
   LOCATION FORMAT [ level ]
   PRIV MODE [=] ON | OFF
   PROMPT
   RADIX [ INPUT | OUTPUT ] [ level ]
   SOURCE { ALL | BACK | FOR | RANGE | WRAP }
   STATUS { ALL | ACTION | LINE25 | SCROLL } [ level ]
   SUBPROC SCOPING [=] [ SUBLOCAL | LOCAL ]
   TRACE { ALL | ARGUMENT[S] | SCOPE[S] | STATEMENT[S] }
level: one of
   BOTH HIGH LOW
```

SOURCE

```
SOURCE [ source-locator ] [ limit-spec ]...
[ file-locator ] [ WRAP ]
source-locator: one of
   AT code-location
   ICODE [ AT code-location ]
   [ LINE ] #line-number
   [ STATEMENT ] statement-number
   SEARCH string [ CASE ] [ position /position ]
limit-spec: one of
   FOR count [ STATEMENT[S] | LINE[S] | INSTRUCTION[S] ]
   BACK count [ STATEMENT[S] | LINE[S] | INSTRUCTION[S] ]
   /position
position: one of
   F L #line-number statement-number
string: one of
   " [ character ]... "
   ' [ character ]... '
file-locator: one of
   FILE file-name
   LOCATION code-location
   SCOPE scope-number
```

STEP

```
STEP [ step-spec ]
step-spec: one of
   num-units [ code-unit ]
   IN [ num-units [ code-unit ] ]
   OUT [ num-calls ] [ PROC[S] | SUBPROC[S] ]
code-spec: one of
   INSTRUCTION[S] STATEMENT[S] VERB[S]
```

STOP

```
STOP [ * | program ]
program: one of
   program-number
   program-name
   cpu,pin
```

SYSTEM

```
SYSTEM [ \system ]
```

TERM

```
TERM { terminal | process }
process:
   [ \system. ] $name
```

TIME

```
TIME [ /OUT/ <file> ]
```

TRACE

```
TRACE [ num-calls ] [ REGISTERS ] [ ARGUMENTS ]
```

VOLUME

```
VOLUME { $volume              }
        { [ $volume. ] subvol }
```

XC

```
XC [ command-line-specifier ]
command-line-specifier: one of
   pos-num
   neg-num
   search-text
   "search-text"
```

Language-Dependent Parameters for C

C Scope Paths

```
scope-path:
   #function
```

C Code Locations

```
code-location:
   { scope-path } [ FROM module°] [ offset°]...
   { [ scope-path.] code-spec }
code-spec: one of
   function
   label
   statement-number
   #line-number [ ( source-file) ]
offset:
   { + | - } num [ code-unit ]
code-unit: one of
   INSTRUCTION[S] STATEMENT[S] VERB[S]
```

C Data Locations

```
data-location:
   [ scope-path [ ( instance) ] . ] data-reference
   [ #data-block. ]
instance:
   [ + | - ] integer
data-reference: one of
   identifier
   data-reference '[' subscript-range ']'
   data-reference.identifier
   data-reference->identifier
   *data-reference
subscript-range:
   expression [ :expression ]
```

C Expressions

expression: one of

primary

**expression*

&*expression*

-*expression*

!*expression*

~*expression*

expression binary-op expression

primary: one of

data-location constant string (*expression*)

binary-op: one of

```
* / % + - >>
<< < > <= >= ==
!= & ^ | && ||
```

Language-Dependent Parameters for C++

C++ Scope Paths

scope-path:

#*function*

C++ Code Locations

```
code-location:
   { scope-path } [ FROM module°] [ offset°]...
   { [ scope-path.] code-spec }
code-spec: one of
   function
   label
   statement-number
   #line-number [ ( source-file) ]
offset:
   { + | - } num [ code-unit ]
code-unit: one of
   INSTRUCTION[S] STATEMENT[S] VERB[S]
```

C ++ Data Locations

```
data-location:
   [ scope-path [ ( instance) ] . ] data-reference
   [ #data-block. ]
instance:
   [ + | - ] integer
data-reference: one of
   identifier
   data-reference '[' subscript-range ']'
   data-reference.identifier
   data-reference->identifier
   *data-reference
subscript-range:
   expression [ :expression ]
```

C++ Expressions

```
expression: one of
   primary
   *expression
   &expression
   -expression
   !expression
   ~expression
   expression binary-op expression
primary: one of
   data-location constant string ( expression )
binary-op: one of
   * / % + - >>
   << < > <= >= ==
   != & ^ | && ||
```

Language-Dependent Parameters for COBOL and SCREEN COBOL

COBOL 74 and SCOBOL Scope Paths

```
scope-path:
   #program-unit
```

COBOL85 Scope Paths

```
scope-path:
   #program-unit [ .program-unit ]...
```

COBOL Code Locations

```
code-location:
   { scope-path } [ offset ]...
   { [ scope-path.] code-spec }
code-spec: one of
   program-unit
   section
   paragraph [ OF section ]
   statement-number
   #line-number [ ( source-file) ]
offset:
   { + | - } num [ code-unit ]
code-unit: one of
   INSTRUCTION[S] STATEMENT[S] VERB[S]
```

COBOL Data Locations

```
data-location:
   [ scope-path. ] data-reference
data-reference:
   identifier [OF identifier ]... [ ( index [ , index ]...) ]
index:
   expression [ :expression ]
```

COBOL Expressions

```
expression:
   condition [ { AND | OR } condition ]...
condition:
   [ NOT ] { simple-exp [ rel-op simple-exp ]... }
           { level-88-condition }
rel-op:
   [ NOT ] { > = < GREATER EQUAL LESS }
simple-exp:
   [ + | - ] term [ { + | - } term ]...
term:
   factor [ { * | / } factor ]...
factor:
   primary [ **primary ]
primary:  one of
   data-location "@data-location" number ( expression )
```

Language-Dependent Parameters for FORTRAN

FORTRAN Scope Paths

```
scope-path:
   #scope-unit
scope-unit:  one of
   program subroutine function
```

FORTRAN Code Locations

code-location:

{ *scope-path* } [*offset*]...

{ [*scope-path*.] *code-spec* }

code-spec: one of

scope-unit

statement-function

statement-label

entry-point

statement-number

#*line-number* [(*source-file*)]

offset:

{ + | - } *num* [*code-unit*]

code-unit: one of

`INSTRUCTION[S] STATEMENT[S] VERB[S]`

FORTRAN Data Locations

data-location:

[*scope-path* [(*instance*)] .] *data-reference*

instance:

[+ | -] *num*

data-reference: one of

identifier

data-reference (*index* [, *index*]...)

data-reference^*identifier*

index:

expression [:*expression*]

FORTRAN Expressions

expression:

    condition [*bool-op condition*]...

bool-op: one of

    .AND. .OR. .EQV. .NEQV.

condition:

    [.NOT.] *simple-exp* [*rel-op simple-exp*]

rel-op: one of

    .LT. .LE. .GT. .GE. .EQ. .NE.

simple-exp:

    [+ | -] *term* [{ + | - } *term*]...

term:

    factor [{ * | / } *factor*]...

factor:

    primary [***primary*]

primary: one of

    data-location constant (*expression*)

Language-Dependent Parameters for Pascal

Pascal Scope Paths

scope-path:

    #*scope-unit* [.*scope-unit*]...

scope-unit: one of

    function procedure

Pascal Code Locations

```
code-location:
   { scope-path } [ FROM module°] [ offset°]...
   { [ scope-path.] code-spec }
code-spec: one of
   scope-unit
   label
   statement-number
   #line-number [ ( source-file) ]
offset:
   { + | - } num [ code-unit ]
code-unit: one of
   INSTRUCTION[S] STATEMENT[S] VERB[S]
```

Pascal Data Locations

```
data-location:
   [ scope-path [ ( instance) ] . ] data-reference
   [ #data-block. ]
instance:
   [ + | - ] integer
data-reference: one of
   identifier
   data-reference '[' index [ , index ]... ']'
   data-reference.identifier
   data-reference^
index:
   expression [ :expression ]
```

Pascal Expressions

expression:

  simple-exp [*rel-op simple-exp*]...

rel-op: one of

  `= <> < > <= >=`

simple-exp:

  [+ | -] *term* [*add-op term*]...

add-op: one of

  `+ - OR`

term:

  factor [*mult-op factor*]...

mult-op: one of

  `* / DIV MOD AND << >>`

factor: one of

  data-location

  unsigned-constant

  NOT *factor*

  (*expression*)

Language-Dependent Parameters for TAL and pTAL

TAL and pTAL Scope Paths

scope-path:

  #procedure

TAL and pTAL Code Locations

```
code-location:
   { scope-path } [ code-offset ]
   { [ scope-path. ] code-reference }
code-reference: one of
   procedure
   subproc
   [ subproc. ] label
   [ subproc. ] entry-point
   statement-number
   #line-number [ ( source-file) ]
code-offset:
   { + | - } num [ code-unit ]
code-unit: one of
   INSTRUCTION[S] STATEMENT[S]
```

TAL and pTAL Data Locations

```
data-location:
   [ scope-path [ ( instance) ] . ]
   [[ scope-path. ] subproc [ ( instance) ]. ] data-reference
   [ #data-block. ]
   [ ##GLOBAL. ]
instance:
   [ + | - ] integer
data-reference: one of
   identifier
   data-reference '[' subscript-range ']'
   data-reference.identifier
subscript-range:
   expression [ :expression ]
```

TAL and pTAL Expressions

```
expression:
   condition [ { AND | OR } condition ]...
condition:
   [ NOT ] simple-exp [ rel-op simple-exp ]...
rel-op: one of
   < <= = >= > <>
   '<' '<=' '=' '>=' '>' '<>'
simple-exp:
   [ + | - ] term [ add-op term ]...
add-op: one of
   + - '+' '-'
   LOR LAND XOR
term:
   factor [ mult-op factor ]...
mult-op: one of
   * / '*' '/' '\'
   << >> '<<' '>>'
factor: one of
   primary
   primary.<primary[:primary]>
primary: one of
   data-location
   .data-location
   @data-location
   number
   ( expression )
```

Low-Level Inspect Commands

Low-level Inspect supports all high-level commands except:

```
BREAK CLEAR DISPLAY MODIFY SCOPE
```

The following low-level Inspect command parameters are described after the low-level commands themselves:

address code-address data-address expr

A

```
A address [ , { count | T entry-size*entry-count } ]
count:
   expr
```

B

B [*code-address* [, *data-address test-op expr*]]

test-op: one of

```
   = < > <> ?
```

BM

BM *address* , *type* [, *data-address test-op expr*]

test-op: one of

```
   = < > <> ?
```

type: one of

```
   R W RW
```

C

C [*code-address*]

CM

```
CM
```

D

```
D [ [ unit ] address [ ,amount ] ] [ :base ]
  [ register ]
```

`unit`: one of

```
   B W D F
```

`amount`: one of

```
   num
   T width * height
```

`base`: one of

```
   A B D H I O X #
```

`register`: one of

```
   P E L S
   R0 R1 R2 R3 R4 R5 R6 R7
   RA RB RC RD RE RF RG RH
```

F

```
F [ expr ]
```

FN

```
FN [ address [ ,value ] [ &mask ] ]
```

HIGH

```
HIGH
```

I

```
I address [ ,amount ]
```

`amount`: one of

```
   num
   T width * height
```

M

```
M { register | data-address } [ ,expr ]...
register:  one of
   P E L S
   R0 R1 R2 R3 R4 R5 R6 R7
   RA RB RC RD RE RF RG RH
```

P

```
P expr
```

R

```
R
```

S

```
S
```

T

```
T [ data-address ]
```

VQ

```
VQ segment-id
```

=

```
= expr [ : display-type ]
display-type:  one of
   A B D H O X #
   C E I
   UC [ .space-id ]
   UL [ .space-id ]
```

?

```
?
```

Low-Level Addresses

address: one of

code-address

data-address

Low-Level Code Addresses

code-address:

[*base-mode*] *expr*

base-mode: one of

`C`

`UC [ .`*space-id* `]`

`UL [ .`*space-id* `]`

Low-Level Data Addresses

data-address:

[*base-mode*] *expr* [*indirection-mode* [*expr*]]

base-mode: one of

`L S Q`

indirection-mode: one of:

`I S IX SX`

Low-Level Expressions

```
expr:
value [ operator value ]...
operator:  one of
   * / << >> + -
value:  one of
   ( expression )
   'ASCII-character ASCII-character
   #code-block
   #data-block
   number [ .number ]
   register
number is:
   [ + | - ] [ # ] integer
register:  one of
   P E L S
   R0 R1 R2 R3 R4 R5 R6 R7
   RA RB RC RD RE RF RG RH
```

C Notes for System Operators

Starting the IMON Process Pair

You should start the IMON process pair as part of your standard system startup procedure. Start $IMON using the command interpreter RUN command.

Note. IMON contains privileged procedures and consequently must be owned by the super id (255,255).

```
[RUN] IMON-program-file /NAME $IMON, [PRI PPP,]
      [CPU nn,] NOWAIT/ bb
```

IMON-program-file

specifies IMON's program file. If you use an implicit RUN command, IMON's program file must be in the $SYSTEM.SYSTEM subvolume or the currently running $SYSTEM.SYS*nn* subvolume. If you enter RUN explicitly, the normal file name expansion rules apply if you specify a partial file name.

NAME $IMON

specifies the process name of the IMON process pair. This run option is required.

PRI PPP

specifies the priority at which $IMON should be run. It is recommended that IMON be started at a relatively high priority.

CPU *nn*

specifies the number of the processor in which the primary IMON process is to run.

bb

specifies the number of the processor in which the backup IMON process is to run.

Default Values

If you do not specify the processor, the primary IMON process runs in the processor where the $CMON process directs it to run. If no $CMON process is present, or if the $CMON process does not specify a processor, the primary IMON process runs in the same processor as the command interpreter from which you entered the RUN command.

If you do not specify a backup processor, the primary IMON process selects a processor for the backup process using these guidelines:

- IMON regards the system as a group of even-odd processor pairs (0,1), (2,3), and so on.
- If the primary IMON process is running in one processor of such a pair, the backup IMON process will be run in the other member of the pair.

Consequently, if your system has an odd number of processors, and the primary IMON process is in the last processor, IMON cannot create a backup because the other member of the pair doesn't exist. In this case, you must specify a backup processor.

Usage Considerations

- IMON assumes that the program files for the DMON and Inspect processes are in the same volume and subvolume as the IMON program file.
- Users can issue an implicit RUN command to start Inspect only if the Inspect program file is in the $SYSTEM.SYSTEM subvolume or the currently running $SYSTEM.SYS*nn* subvolume.

IMON and CMON

The IMON process always communicates with the CMON process (if present) for the CPU number and priority to start Inspect. Inspect should be started at the same priority as command interpreters or other interactive programs. The IMON process sends the request to the CMON process with the Inspect file name in the local internal file format, which implies that the Inspect file name does not contain the node number.

Note. If the CMON process is not present or the IMON process is unable to communicate with the CMON process, the IMON process starts Inspect at its own priority.

Stopping IMON and DMON Processes

To stop the Inspect subsystem, stop the IMON process and then stop all DMON processes. To stop IMON, use the name form of the TACL command STOP. For example:

```
> STOP $IMON
```

Only the super ID (255,255) user can stop the primary IMON process.

Stopping the $IMON process pair *by name* stops only the IMON processes, not the DMON processes. You should stop all the DMON processes after a $IMON pair has been stopped. No special shutdown procedures are required for IMON and DMON processes.

IMON restarts DMON processes when DMON ABENDs or STOPs. If DMON repeatedly ABENDs, IMON will not restart DMON, and issue an error message. To restart DMON, you must restart IMON.

IMON and DMON Errors

Errors occurring during IMON startup are displayed on the home terminal. Internal IMON and DMON errors that occur during operation are reported on the operator console. Errors related to a user processes are sent to the process's home terminal.

DMON and IMON error messages are displayed in this format:

```
*** IMON ERROR *** message

*** $DMnn ERROR *** message
```

where message describes the particular error and nn identifies an individual DMON process.

Errors Common to IMON and DMON

```
Internal error at P=%nnnnnn.
```

This message is reported when the IMON process or one of the DMON processes is recovering from an internal error. It should be ignored, unless the users of Inspect are experiencing problems; in this case, the error should be reported to your Tandem representative.

IMON Errors

```
IMON must be named $IMON
```

You omitted the NAME $IMON option from the RUN command.

```
IMON must be started by the Super ID
```

IMON must be started by user id 255,255.

```
PROCESS_CREATE_ error: nn error detail: nn
Filename: file
```

Process creation failed for the reason indicated by the error number. The program file name is displayed. This message will appear on the user's terminal if a debug event triggered an attempt by IMON to start an Inspect process, and PROCESS_CREATE_ was unable to comply. It will appear on the operator console if an IMON process attempted to start the a DMON process and PROCESS_CREATE_ was unable to comply.

```
PROCESS_GETINFO_ error: nn error detail: nn
on cpu,pin
```

IMON will print this out when the procedure PROCESS_GETINFO_ returns an error.

```
Wrong Guardian version
```

You tried to run the program with an incompatible version of the operating system.

```
DMON in CPU nn marked down by $IMON
```

This message is reported when DMON repeatedly ABENDs or STOPs. IMON will mark it as down, and will not attempt to restart it. IMON must be re-started to mark this DMON up again.

```
IMON must be named $IMON
```

The NAME $IMON option was omitted from the RUN command.

```
Backup creation loop
```

IMON is in a loop trying to create its backup. IMON will stop itself.

```
INSPECT command terminal name cannot be converted to internal
format
```

Inspect terminal name cannot be converted into internal format.

```
Illegal startup parameter
```

You specified an invalid CPU in the RUN command.

```
$RECEIVE - error-text
```

IMON will print this out when an error on $RECEIVE occurs.

```
INSPECT - error-text
```

There was an error opening or writing to an Inspect process.

```
DMON - error-text
```

There was an error opening or writing to a DMON process.

```
Checkpoint with backup failed. [error-text]
```

A checkpoint with IMON's backup failed.

DMON Errors

The following errors are reported by DMON.

```
Initialization of Guardian event reporting failed
```

This error is reported at DMON start up time if, for some reason, the initialization of the operating system mechanism that reports debugging events to DMON failed.

```
IMON ($IMON) does not exist
```

This error is reported if DMON receives a debugging event and no IMON process exists.

```
Fatal trap %n at P = %xxxxxx
```

This error is reported if the DMON process encounters a run-time trap. n is the trap number (in octal), and xxxxxx is the address.

```
DMON must be started by $IMON
```

This error is reported if an attempt is made to start DMON by running the DMON program file. To start DMON, you must start IMON, IMON then starts a DMON in each CPU.

```
Process deleted during DMON processing: process
```

This error is reported if, when DMON is processing a debugging event, it finds that the process no longer exists. Process is a string of the form cpu,pin [(name)]. For example, 1,30 ($DBMON).

```
Save file error nn, volume vol, process id, object file
```

DMON reports this error if an error occurs during the creation of a save file for a process. Depending on the error, this could indicate a resource shortage, such as disk space.

```
Save file error - internal, P-%nnnnnn, process id, object
file
```

This error is reported if an internal error is detected. If this error occurs, contact your Tandem representative.

```
Unable to stop process: process
```

This error is reported if DMON's attempt to stop a process fails. (when a process that has its Inspect bit set stops or abends, control of the process is given to DMON, allowing it to create a save file if appropriate. DMON then calls STOP to complete stopping the process. This error is reported if the STOP operation fails). When this event occurs, the process still exists.

```
ALLOCATESEGEMENT failed: Volume volume, Error error
```

This error is reported when DMON begins execution if its attempt to allocate an extended data segment fails.

```
Incompatible GUARDIAN version
```

User tried to run DMON on a incompatible version of the operating system.

```
SAFEGUARD error nnn on $ZSMP, save file might retain
SUPER.SUPER default protection.
```

DMON reports this error when an error occurs during the communication to SAFEGUARD ($ZSMP).

```
SAFEGUARD error nnn (SPI) from $ZSMP, save file might retain
SUPER.SUPER default protection.
```

DMON reports this error when an SPI error occurs during the SAFEGUARD operations on a save file being created.

```
Process name CPU,PIN is a TNS/E process and requires the use
of Native Inspect or Visual Inspect. Inspect does not support
TNS/E native processes.
```

IMON reports this error when you try to invoke Inspect to debug a TNS/E native process. To transfer the process to Visual Inspect,:

1. Start the Visual Inspect client and connect to the NonStop system.
2. From the Visual Inspect client, select the Open Program command and specify the process name or CPU, PIN.

At the Native Inspect prompt, enter the switch command.

Glossary

accelerate. To speed up emulated execution of a TNS object file by applying either the Axcel accelerator for TNS/R system execution or the Object Code Accelerator (OCA) for TNS/E system execution before running the object file.

accelerated mode. See TNS accelerated mode.

accelerated object code. Either the MIPS RISC instructions (in the MIPS region) that result from processing a TNS object file with the Axcel accelerator on a TNS/R system or the Intel® Itanium® instructions (in the Itanium instruction region) that result from processing a TNS object file with the TNS Object Code Accelerator (OCA) on a TNS/E system.

accelerated object file. A TNS object file that contains accelerated object code. An accelerated object file can contain both an Axcel region and an Itanium instruction region.

Accelerator. A program optimization tool that processes a TNS object file and produces an accelerated object file that contains equivalent native instructions. On TNS/R systems, the accelerator is named Axcel. On TNS/E systems, the accelerator is the TNS Object Code Accelerator (OCA),

Accelerator mode. An operational environment in which an object file executes accelerated code, as follows:

- On TNS/E systems, Itanium instructions are executed that were generated by the Object Code Accelerator (OCA)
- On TNS/R systems, RISC instructions are executed that were generated by the Axcel accelerator

Contrast with TNS/R native mode. See also TNS mode and TNS/R native mode.

Accelerator region of an object file. The region (called the MIPS region on TNS/R systems, and the Itanium region on TNS/E systems) of an object file that contains instructions and tables necessary to execute the object file on a TNS/R or TNS/E system in accelerated mode. The accelerator creates this region. Contrast with OCA region of an object file. See also accelerated object file.

Active scope units. Any scope unit that has been called but has not yet been exited. The call history displayed by the TRACE command lists all active scope units.

Banner line. The line of information Inspect displays when you begin an Inspect session.

Break action. The part of a breakpoint's definition that specifies what to do after the breakpoint triggers a break event.

Break condition. The part of a breakpoint's definition that specifies under what conditions the breakpoint is to trigger a break event.

Break event. The type of debug event that occurs when a breakpoint causes a program's execution to halt.

Break location. The part of a breakpoint's definition that specifies where the breakpoint is located.

Breakpoint. A location (or point) in a program where execution is to be suspended so that you can then examine and perhaps modify the program's state. You can set and clear breakpoints with Inspect commands. In this manual, a breakpoint is assumed to be in TNS memory unless the location is called a "TNS/R breakpoint."

Breakpoint activation. The time at which program execution encounters a breakpoint. Breakpoint activation results in a break event if the breakpoint has no conditions or if its conditions are met.

Breakpoint definition. The time at which you set a breakpoint (using the BREAK command).

Breakpoint list. The list of breakpoints defined in a program. When debugging several programs concurrently, Inspect maintains a separate breakpoint list for each program.

Call history. The list of scope-unit activations (calls) displayed by the TRACE command. TRACE orders this list from the most recent to the earliest call.

Clause. An extension to an Inspect command. All command clauses start with or include a descriptive keyword that identifies the clause.

CISC. See complex instruction-set computing (CISC).

Code base. The address at which the code for a scope unit begins—not to be confused with the primary entry point, which is the address at which execution of a scope unit begins.

Code block. The smallest cluster of object code that BINDER can relocate separately from the rest of the object code. BINDER code blocks and Inspect scope units are roughly equivalent, but they are not interchangeable terms.

Code breakpoint. A breakpoint set at a code location.

Code location. A specific location within a program's object code. Inspect lets you define code locations using source-language expressions and identifiers in addition to machine-level addresses and offsets.

Code offset. An offset from a named code location. Code offsets are specified as a number of code units.

Code unit. A measure of code. There are three code units: INSTRUCTION, STATEMENT, and VERB.

Command list. A list of one or more Inspect commands, separated by semicolons.

Command mode. An operating mode of Inspect that defines the type of access you have to your program. The low-level command mode provides machine-level access, and high-level command mode provides source-level access.

Command terminal. A terminal at which system managers, developers, and programmers interact with system-level utility programs.

Common Run-Time Environment (CRE). A set of services implemented by the CRE library that supports mixed-language programs on D-series systems. Contrast with language-specific run-time environment.

Common Run-Time Environment (CRE) library. A collection of routines that supports requests for services managed by the CRE, such as I/O and heap management, math and string functions, exception handling, and error reporting. CRE library routines can be called by C, COBOL85, FORTRAN, Pascal, and TAL user routines and run-time libraries.

compiler extended-data segment. A selectable segment, with ID 1024, created and selected automatically in many (but not all) TNS processes. Within this segment, the compiler automatically allocates global and local variables and heaps that would not fit in the TNS user data segment. A programmer must keep this segment selected whenever those items might be referenced. Any alternative selections of segments must be temporary and undone before returning.

complex instruction-set computing (CISC). A processor architecture based on a large instruction set, characterized by numerous addressing modes, multicycle machine instructions, and many special-purpose instructions. Contrast with reduced instruction-set computing (RISC).

conditional breakpoint. A breakpoint defined with break conditions that govern its ability to trigger a break event.

CRE. See Common Run-Time Environment (CRE).

current breakpoint. The breakpoint that caused the most recent break event in a program, and beyond which that program's execution has not advanced.

current program. The program that you are currently debugging. If you are debugging several programs concurrently, you use the PROGRAM command to specify which of them is the current program.

current scope path. The scope path that Inspect uses to complete any partially qualified code or data references. Whenever a debug event occurs in a program you are debugging, Inspect automatically sets the current scope path to the path that defines the scope unit where the event occurred. You can set the current scope path explicitly using the SCOPE command.

data block. A collection of static data items that BINDER can relocate separately from the rest of a program's static data. Dynamic data items (local variables, file buffers, and so on) are not part of any data block.

data breakpoint. A breakpoint set at a data location.

data location. A specific location within a program's static or dynamic data area. Inspect lets you define data locations using source-language symbols and identifiers.

Debug event. Any event that forces a program into the hold state.

debugging attributes. Two attributes of a process or PATHWAY server. The Inspect attribute determines which debugging tool (Inspect or Debug) should respond to a debug event, and the SAVEABEND attribute determines whether the program state is recorded in a save file if the program terminates abnormally (abends).

debugging session. Another name for an Inspect session.

DMON process; DMON, $DMnn. The process that enables Inspect to access processes and PATHWAY servers. Its duties include setting and clearing breakpoints, storing and retrieving data, informing the Inspect process when a breakpoint is encountered, and creating save files.

D-series system. . An HP NonStop system that is running a D-series version of the operating system.

dynamic-link library (DLL). A collection of procedures whose code and data can be loaded and executed at any virtual memory address, with run-time resolution of links to and from the main program and other independent libraries. The same DLL can be used by more than one process. Each process gets its own copy of DLL static data. Contrast with shared run-time library (SRL). See also TNS/R library.

eld utility. The utility that collects, links, and modifies code blocks and data blocks from one or more object files to produce a target TNS/E native object file with shared code (PIC). Compare to nld utility and ld utility.

enoft utility. The utility that reads and displays information from TNS/E native object files. Compare to noft utility.

execution states. The phases into which Inspect groups the execution of a program. These states describe the current activity of a program and determine the availability and significance of certain Inspect commands. The three execution states are: the run state, the hold state, and the stop state.

Explicitly Parallel Instruction Computing (EPIC). The technology that forms the basis for the Intel Itanium architecture of the TNS/E system. EPIC technology allows parallel processing opportunities to be explicitly identified by the compiler before the software code is executed by the processor.

expression. A list of operands and operators which, when evaluated, results in a number or a string.

gone state. A state where the program has been stopped, but the program will not be cleared from the list until DMON sends a message back. When the program is in this state it is possible to catch it before it actually stops.

G-series system. .An HP NonStop system that is running a G-series version of the operating system.

Guardian. An environment available for interactive or programmatic use with the NonStop operating system. Processes that run in the Guardian environment use the system procedure calls as their application program interface; interactive users of the Guardian environment use the HP Tandem Advanced Command Language (TACL) or another product's command interpreter. Contrast with Open System Services (OSS).

high-level mode; high-level Inspect. The Inspect command mode that provides source-level access to your program.

high PIN. A process identification number (PIN) that is greater than 255. Contrast with low PIN.

Hold state. The execution state describing a program that has been suspended temporarily, but can resume execution again.

H-series system. .An HP NonStop system that is running an H-series version of the operating system.

home terminal. The terminal from which a process is started.

identifier. A name that symbolically identifies an object. Most programming languages use identifiers to define data and code items.

IMON process pair; IMON, $IMON. The fault-tolerant process pair that manages and monitors all Inspect sessions on a single system. Its duties include starting and maintaining DMON processes on each of the CPUs in the system, setting up the link between a DMON process and an Inspect process, and creating Inspect processes when necessary.

implicit library or DLL. A library or DLL supplied by HP that is available in the read-only and execute-only globally mapped address space shared by all processes without being specified to the linker or loader. See also TNS system library and public library.

inactive scope unit. Any scope unit that does not have a call in effect; that is, a scope unit that does not appear in the call history.

Inspect command terminal. The terminal by which Inspect communicates with you during an Inspect session.

Inspect process. The terminal process that enables you to debug programs distributed across several DMONs and TCPs. In addition, the Inspect process enables Inspect to access symbol information and source code for programs written in all languages.

Inspect prompt. The prompt that Inspect displays when awaiting your command. See Prompting for Commands on page 2-3.

Inspect region. The region of a TNS object file that contains symbol tables for all blocks compiled with the SYMBOLS directive. The Inspect region is sometimes called the symbols region.

INSPSNAP. The program that provides a process snapshot file for the Inspect subsystem.

Inspect session. The time period during which you debug interactively using the same Inspect process. An Inspect session begins when IMON first creates an Inspect process on your terminal, and continues until you terminate that process.

instance. A specific activation of a scope unit. If a scope unit calls itself, several activations of that scope unit will appear in the call history. Each such activation is an instance.

Intel® Itanium® instructions. Register-oriented Itanium instructions that are native to and directly executed by a TNS/E system. Itanium instructions do not execute on TNS and TNS/R systems. Contrast with TNS instructions.

The Object Code Accelerator (OCA) produces Itanium instructions to accelerate TNS object code. A TNS/E native compiler produces native-compiled Itanium instructions when it compiles source code.

Intel Itanium instruction region. The region of a TNS object file that contains Itanium instructions and the tables necessary to execute the instructions in accelerated mode on a TNS/E system. The Object Code Accelerator (OCA) creates this region and writes it into the TNS object file. Synonym for OCA region of an object file. Contrast with MIPS region of a TNS object file

Intel Itanium word. An instruction-set-defined unit of memory. An Itanium word is 4 bytes (32 bits) wide, beginning on any 4-byte boundary in memory. Contrast with TNS word and word. See also MIPS RISC word.

interpreted mode. See TNS interpreted mode.

ld utility. The utility (also called the native PIC linker) that collects, links, and modifies code blocks and data blocks from one or more object files to produce a target TNS/R native object file with shared code (PIC). Compare to nld utility and eld utility. Compare to nld utility and ld utility

linker. A system utility that collects, links, and modifies code and data blocks from one or more object files to produce a target object file or loadfile.

Four linkers are available on NonStop systems:

- Binder links TNS object files on TNS/R systems.
- `nld` links TNS/R native object files to create a linkfile or a loadfile.
- `ld`, the PIC (Position-Independent Code) linker, links TNS/R PIC object files to create a PIC executable, which can be either a program file or a dynamic-link library (DLL).
- `eld`, the TNS/E linker, links TNS/E object files to a PIC executable, which can be either a program file or a dynamic-link library (DLL).

linkfile. A linkable object file produced by the compiler from a source code file. Also, a command file used for input to a linker. See also loadfile.

linking. The operation of examining, collecting, linking, and modifying code and data blocks from one or more object files to produce a target object file or loadfile.

loadfile. An executable file that is ready for loading and executing on the HP NonStop server. Examples are programs, user library files, SRLs and DLLs. Compare with object file.

There are two basic types of loadfiles:

- Programs, which define an entry point where execution of the program begins
- Libraries, which supply functions or data to a client loadfile

low-level mode; low-level Inspect. The Inspect command mode that provides machine-level access to your program.

low PIN. A process identification number (PIN) in the range 0 through 254. Contrast with high PIN.

machine (language) instruction. The low-level instructions produced when you compile programs written in C, COBOL, FORTRAN, Pascal, or TAL. A CPU executes these instructions.

memory-exact point. A potential breakpoint location within an accelerated object file at which the values in memory (but not necessarily the values in registers) are the same as they would be if the object file were running in TNS interpreted mode or on a TNS system. Most source statement boundaries are memory-exact points. Complex statements might contain several such points: at each function call, privileged instruction, and embedded assignment. Contrast with register-exact point and nonexact point.

millicode. TNS/R instructions that implement various TNS low-level functions such as exception handling, real-time translation routines, and library routines that implement the TNS instruction set. TNS/R millicode is functionally equivalent to TNS microcode.

MIPS region of a TNS object file. The region of a TNS object file that contains MIPS instructions and the tables necessary to execute the instructions in accelerator mode on a TNS/R system. The accelerator creates this region and writes it into the TNS object file. Contrast with Intel Itanium instruction region.

MIPS RISC word. An instruction-set-defined unit of memory. A MIPS RISC word is 4 bytes (32 bits) wide, beginning on any 4-byte boundary in memory. Contrast with TNS word and word. See also Intel Itanium word.

native. An adjective that can modify object code, object file, process, procedure, and mode of process execution. Native object files contain native object code, which directly uses either MIPS or Intel Itanium instructions and the corresponding conventions for register handling and procedure calls. Native processes are those created by executing native object files. Native procedures are units of native object code. Native mode execution is the state of the process when it is executing native procedures.

native mode. See TNS/R native mode and TNS/E native mode.

native object code. See TNS/R native object code and TNS/E native object code.

native object file. See TNS/R native object file and TNS/E native object file.

native object file tool. See noft utility and enoft utility,

native process. See TNS/R native process and TNS/E native process.

native system library or DLL. Synonym for implicit library or DLL.

nld utility. A utility that collects, links, and modifies code and data blocks from one or more object files to produce a target TNS/R native object file. The nld utility is similar to the Binder program used in the TNS development environment. Compare to eld utility.

noft utility. A utility that reads and displays information from TNS/R native object files. Compare to enoft utility

nonexact point. A code location within an accelerated object file that is between memory-exact points. The mapping between the TNS program counter and corresponding RISC instructions is only approximate at nonexact points, and interim changes to memory might have been completed out of order. Breakpoints cannot be applied at nonexact points. Contrast with memory-exact point and register-exact point.

NSR-L processor. The NonStop System RISC Model L processor (NSR-L processor) is the first TNS/R processor. NonStop Cyclone/R and CLX 2000 systems contain NSR-L processors. All documentation that refers to the NSR-L processor and related software applies to both systems unless explicitly stated otherwise.

object file. A file, generated by a compiler or binder, that contains machine instructions and other information needed to construct the code spaces and initial data for a process.

object code accelerator (OCA). See TNS Object Code Accelerator (OCA).

OCA. (1) The command used to invoke the TNS Object Code Accelerator (OCA) on a TNS/E system. (2) See TNS Object Code Accelerator (OCA).

OCA region loading. A task performed when necessary by the TNS emulation software for TNS/E machines. This task involves mapping into memory the Intel Itanium instructions and any tables needed at run time from the OCA-generated object file.

OCA region of an object file. The region of a Object Code Accelerator (OCA)-generated object file, also called the Intel Itanium instruction region, that contains Itanium instructions and tables necessary to execute the object file on a TNS/E system in TNS accelerated mode. The Object Code Accelerator (OCA) creates this region. See also OCA-accelerated object code. Contrast with Accelerator region of an object file.

OCA-accelerated object code. The Intel Itanium instructions that result from processing a TNS object file with the Object Code Accelerator (OCA).

OCA-accelerated object file. A TNS object file that has been augmented by the TNS Object Code Accelerator (OCA) with equivalent but faster Intel Itanium instructions. An OCA-accelerated object file contains the original TNS object code, the OCA-accelerated object code and related address map tables, and any Binder and symbol information from the original TNS object file. An OCA-accelerated object file also can be augmented by the Axcel accelerator with equivalent MIPS RISC instructions.

OCA-generated Itanium instructions. See Intel® Itanium® instructions.

open file. A file with a file descriptor.

open file description. A data structure within a NonStop node that contains information about the access of a process or of a group of processes to a file. An open file description records such attributes as the file offset, file status, and file access modes. An open file description is associated with only one open file but can be associated with one or more file descriptors.

Open System Services (OSS). An open system environment available for interactive or programmatic use with the HP NonStop operating system. Processes that run in the OSS environment usually use the OSS application program interface. Interactive users of the OSS environment usually use the OSS shell for their command interpreter. Synonymous with *Open System Services (OSS) environment*. Contrast with Guardian.

OSS. See Open System Services (OSS)

OSS process ID (PID). The unique identifier that represents a process during the lifetime of the process and during the lifetime of the process group of that process. See also “PID.”

OSS signal. A signal model defined in the POSIX.1 specification and available to TNS processes and TNS/R native processes in the OSS environment. OSS signals can be sent between processes.

parameter. An item you specify in an Inspect command. Parameters always appear in italic print in the manual.

pathname. The string of characters that uniquely identifies a file within its file system. A pathname can be either relative or absolute. See also ISO/IEC IS 9945-1:1990 (ANSI/IEEE Std. 1003.1-1990 or POSIX.1), Clause 2.2.2.57.

PATHWAY requester program. A SCREEN COBOL program running under the supervision of a TCP. A PATHWAY requester is composed of a PATHWAY requester program and the TCP supervising it.

PATHWAY server. An application process that accepts a request from a PATHWAY requester, fulfills the request, and returns a reply to the requester.

pending state. .The execution state where you don’t see the program while it is running.

PIC (Position-Independent Code). Executable code that can run at different virtual addresses.

PID. In the OSS environment, PID stands for OSS process ID, a numeric identifier assigned to an OSS process and unique within a NonStop node.

In the Guardian environment, PID is used to mean either of these:

- A Guardian process identifier such as the process ID
- The cpu, pin value that is unique to a process within a NonStop node

  See also OSS process ID (PID).

PIN. See “Process identification number.”

primary entry point. The address at which execution of a scope unit begins—not to be confused with code base, which is the address at which the code for a scope unit begins.

ProcDebug. An accelerator option that directs the accelerator to perform optimization across statement boundaries. This option typically produces faster-executing code than the StmtDebug option, but debugging the program might be more difficult because it might not be possible to set a breakpoint at some statement boundaries. ProcDebug is the accelerator default. Contrast with StmtDebug.

process. A running machine-code program.

process handle. A system data structure that serves as the address of a named or unnamed process in the network. A process handle identifies an individual process, not a process pair.

process ID. A system data structure that serves as a address of a process. The structure contains a CPU number, PIN, creation timestamp or process name, and system number (optional). It is sometimes called a creation timestamp process ID (CRTPID).

process identification number (PIN). An unsigned integer that identifies a process in a processor module.

process snapshot file. (1) A file containing dump information needed by the system debugging tool. In UNIX systems, such files are usually called core files or core dump files. (2) A file containing the state of a running process at a specific moment, regardless of whether the process terminated abnormally. See also saveabend file.

program. In the general sense, a program is a file that contains a series of instructions. In Inspect, however, the term “program” has a more specific meaning—a program is a process, a save file, a PATHWAY server, or a PATHWAY requester program.

program file. An executable object file generated by a compiler or by the accelerator and a binder program.

program list. The list of programs that you are debugging concurrently. The PROGRAM command without any parameters displays the program list.

pseudocode. The intermediate-level code produced when you compile a SCREEN COBOL program. A TCP, not a CPU, executes pseudocode.

public library. A dynamic-link library (DLL) or shared run-time library (SRL) that is known to the operating system, available for execution by any process or user, and is not an implicit library.

reduced instruction-set computing (RISC). A processor architecture based on a relatively small and simple instruction set, a large number of general-purpose registers, and an optimized instruction pipeline that supports high-performance instruction execution. Contrast with complex instruction-set computing (CISC) and Explicitly Parallel Instruction Computing (EPIC).

register-exact point. A synchronization location within an accelerated object file at which both of these statements are true:

- All live TNS registers plus all values in memory are the same as they would be if the object file were running in TNS mode or TNS interpreted mode or on a TNS system.
- All accelerator code optimizations are ended.

Register-exact points are a small subset of all memory-exact points. Procedure entry and exit locations and call-return sites are usually register-exact points. All places where the program might switch into or from TNS mode or TNS interpreted mode are register-exact points. Contrast with memory-exact point and nonexact point.

RISC. See “reduced instruction-set computing (RISC).”

RISC instructions. Register-oriented 32-bit machine instructions that are directly executed on TNS/R processors. RISC instructions execute only on TNS/R systems. Contrast with TNS instructions and Intel® Itanium® instructions.

RISC processor. An instruction processing unit (IPU) that is based on reduced instruction-set computing (RISC) architecture. TNS/R systems contain RISC processors.

run state. The execution state describing a program while it is running.

save file. A file created by the Inspect subsystem in response to a command from a debugger. A save file contains enough information about a running process at a given time to restart the process at the same point in its execution. A save file contains an image of the process, data for the process, and the status of the process at the time the save file was created.

A save file can be created through an Inspect SAVE command at any time. A save file called a saveabend file can be created by DMON if a process's SAVEABEND attribute is set and the process terminates abnormally. Other debuggers can create a save file but refer to the result as a process snapshot file. See also process snapshot file.

saveabend file. A file containing dump information needed by the system debugging tool. (In UNIX systems, such files are usually called core files or core dump files.) A saveabend file is a special case of a save file. See also save file and process snapshot file.

scope of identifiers; scope. The concept that an identifier in a source program has a defined range, or domain, over which it is valid.

scope path. The list of scope units containing a source identifier, starting with the outermost and working down to the innermost (the one immediately containing the identifier).

scope units; scope unit. The entity or entities into which source code is grouped. Scope units enable programmers to define the boundaries of scope domains.

Shared Millicode Library. An intrinsic library containing privileged or TNS-derived millicode routines used by many native-compiled programs and by emulated TNS programs. This library includes efficient string-move operations, TNS floating-point emulation, and various privileged-only operations. These routines are mode independent. They comply with native calling conventions but can be directly invoked from any mode without changing execution modes.

shared run-time library (SRL). A collection of procedures whose code and data can be loaded and executed only at a specific assigned virtual memory address (the same address in all processes). SRLs use direct addressing and do not have run-time resolution of links to and from the main program and other independent libraries. Contrast with dynamic-link library (DLL). See also TNS shared run-time library (TNS SRL) and TNS/R native shared run-time library (TNS/R native SRL).

signal. A means by which a process can be notified of or affected by an event occurring in the system. Signals are used to notify a process when an error not related to input or output has occurred. See also OSS signal, TNS signal, TNS/R native signal, and TNS/E native signal.

signal handler. A procedure that is executed when a signal is received by a process.

snapshot. An image of the process, its data, and its status at the moment it was saved. In Inspect, a snapshot, referred to as a save file, is created using the SAVE command. In Visual Inspect, a snapshot is created using the Save Snapshot command. If the SAVEABEND attribute for a process is ON and the process abends, the snapshot file server INSPSNAP creates a snapshot (file code 130). Both Visual Inspect and Inspect can be used to debug snapshots.

SRL. See “shared run-time library (SRL).”

status message. The informational message that Inspect displays when a debug event occurs.

StmtDebug. An accelerator option that directs the accelerator to optimize instructions only within the code produced for any one statement. Instructions are not optimized across statements. This option typically produces less-optimized code than the ProcDebug option. However, debugging is easier than with the ProcDebug option because the beginning of every statement in the source program is a memory-exact point. Contrast with ProcDebug.

Stop state. The execution state describing a program immediately after it completes execution (whether normally or abnormally).

super ID. On HP NonStop systems, a privileged user who can read, write, execute, and purge all files on the system. The super ID is usually a member of a system-supervisor group.

The super ID has the set of special permissions called appropriate privileges. In the Guardian environment, the structured view of the super ID, which is (255, 255), is most commonly used. In the Open System Services (OSS) environment, the scalar view of the super ID, which is 65535, is most commonly used.

symbol. (The symbolic name of a value, typically a function entry point or a data location. In the context of loadable libraries, symbols are defined in loadfiles and referenced in the same or other loadfiles.

symbols region. See Inspect region.

TCP. A process that is responsible for managing PATHWAY application terminals and executing the pseudo-instructions in PATHWAY requester programs. A PATHWAY requester is composed of a PATHWAY requester program and the TCP executing it.

TNS. Refers to fault-tolerant HP computers that support the HP NonStop operating system and are based on microcoded complex instruction-set computing (CISC) technology. TNS systems run the TNS instruction set. Contrast with TNS/R and TNS/E.

TNS accelerated mode. A TNS emulation environment on a TNS/R or TNS/E system in which accelerated TNS object files are run. TNS instructions have been previously translated into optimized sequences of MIPS or Intel Itanium instructions. TNS accelerated mode runs much faster than TNS interpreted mode. Accelerated or interpreted TNS object code cannot be mixed with or called by native mode object code. See also Accelerator and TNS Object Code Accelerator (OCA). Contrast with TNS interpreted mode, TNS/R native mode and TNS/E native mode.

TNS C compiler. The C compiler that generates TNS object files. Contrast with TNS/R native C compiler and TNS/E native C compiler.

TNS code segment. One of up to 32 128-kilobyte areas of TNS object code within a TNS code space. Each segment contains the TNS instructions for up to 510 complete routines. Each TNS code segment contains its own procedure entry-point (PEP) table and external entry-point (XEP) table. It can also contain read-only data.

TNS code segment identifier. A seven-bit value in which the most significant two bits encode a code space (user code, user library, system code, or system library) and the five remaining bits encode a code segment index in the range 0 through 31.

TNS code segment index. A value in the range 0 through 31 that indexes a code segment within the current user code, user library, system code, or system library space. This value can be encoded in five bits.

TNS code space. One of four addressable collections of TNS object code in a TNS process. They are User Code (UC), User Library (UL), System Code (SC), and System Library (SL). UC and UL exist on a per-process basis. SC and SL exist on a per-node basis.

TNS compiler. A compiler in the TNS development environment that generates 16-bit TNS object code following the TNS conventions for memory, stacks, 16-bit registers, and call linkage. The TNS C compiler is an example of such a compiler. Contrast with TNS/R native compiler and TNS/E native compiler.

TNS Emulation Library. A public dynamic-link library (DLL) on a TNS/E system containing the TNS object code interpreter, millicode routines used only by accelerated mode, and millicode for switching among interpreted, accelerated, and native execution modes. See also Shared Millicode Library.

TNS emulation software. The set of tools, libraries, and system services for running TNS object code on TNS/E systems and TNS/R systems. On a TNS/E system, the TNS emulation software includes the TNS object code interpreter, the TNS Object Code Accelerator (OCA), and various millicode libraries. On a TNS/R system, the TNS emulation software includes the TNS object code interpreter, the Axcel accelerator, and various millicode libraries.

TNS fixup. A task performed at process startup time when executing a TNS object file. This task involves building the procedure entry point (PEP) table and external entry point

(XEP) table and patching PCAL and XCAL instructions in a TNS object file before loading the file into memory. See also TNS mode.

TNS instructions. Stack-oriented, 16-bit machine instructions that are directly executed on TNS systems by hardware and microcode. TNS instructions can be emulated on TNS/E and TNS/R systems by using millicode, an interpreter, and acceleration. Contrast with Intel® Itanium® instructions and RISC instructions.

TNS interpreted mode. A TNS emulation environment on a TNS/R or TNS/E system in which individual TNS instructions in a TNS object file are directly executed by interpretation rather than permanently translated into MIPS or Intel Itanium instructions. TNS interpreted mode runs slower than TNS accelerated mode. Each TNS instruction is decoded each time it is executed, and no optimizations between TNS instructions are possible. TNS interpreted mode is used when a TNS object file has not been accelerated for that hardware system, and it is also sometimes used for brief periods within accelerated object files. Accelerated or interpreted TNS object code cannot be mixed with or called by native mode object code. Contrast with TNS accelerated mode, TNS/R native mode, and TNS/E native mode.

TNS library. A single, optional, TNS-compiled loadfile associated with one or more application loadfiles. If a user library has its own global or static variables, it is called a TNS shared run-time library (TNS SRL). Otherwise it is called a User Library (UL).

TNS loading. A task performed at process startup time when executing a TNS object file. This task involves mapping the TNS instructions, procedure entry-point (PEP) table, and external entry-point (XEP) table from a TNS object file into memory.

TNS mode. The operational environment in which TNS instructions execute by inline interpretation. See also accelerated mode, TNS/R native mode and TNS interpreted mode.

TNS object code. The TNS instructions that result from processing program source code with a TNS compiler. TNS object code executes on TNS, TNS/R, and TNS/E systems.

TNS Object Code Accelerator (OCA). A program optimization tool that processes a TNS object file and produces an accelerated file for a TNS/E system. OCA augments a TNS object file with equivalent Intel Itanium instructions. TNS object code that is accelerated runs faster on TNS/E systems than TNS object code that is not accelerated. See also Accelerator and TNS Object Code Interpreter (OCI).

TNS Object Code Interpreter (OCI). A program that processes a TNS object file and emulates TNS instructions on a TNS/E system without preprocessing the object file. See also TNS Object Code Accelerator (OCA).

TNS object file. An object file created by a TNS compiler or the Binder. A TNS object file contains TNS instructions. TNS object files can be processed by the Axcel accelerator or by the TNS Object Code Accelerator (OCA) to produce to produce accelerated object files. A TNS object file can be run on TNS, TNS/R, and TNS/E systems.

TNS procedure label. A 16-bit identifier for an internal or external procedure used by the TNS object code of a TNS process. The most-significant 7 bits are a TNS code segment identifier: 2 bits for the TNS code space and 5 bits for the TNS code segment index. The least-significant 9 bits are an index into the target segment's procedure entry-point (PEP) table. On a TNS/E system, all shells for calling native library procedures are segregated within the system code space. When the TNS code space bits of a TNS procedure label are %B10, the remaining 14 bits are an index into the system's shell map table, not a segment index and PEP index.

TNS process. A process whose main program object file is a TNS object file, compiled using a TNS compiler. A TNS process executes in interpreted or accelerated mode while within itself, when calling a user library, or when calling into TNS system libraries. A TNS process temporarily executes in native mode when calling into native-compiled parts of the system library. Object files within a TNS process might be accelerated or not, with automatic switching between accelerated and interpreted modes on calls and returns between those parts. Contrast with TNS/R native process and TNS/E native process.

TNS shared run-time library (TNS SRL). A shared run-time library (SRL) available to TNS processes in the Open System Services (OSS) environment. A TNS process can have only one TNS SRL. A TNS SRL is implemented as a special user library that allows shared global data. See also shared run-time library (SRL) and dynamic-link library (DLL).

TNS signal. A signal model available to TNS processes in the Guardian environment.

TNS stack segment. See TNS user data segment.

TNS State Library for TNS/E. A library of routines to access and modify the TNS state of a TNS process running on TNS/E.

TNS system library. A collection of HP-supplied TNS-compiled routines available to all TNS processes. There is no per-program or per-process customization of this library. All routines are immediately available to a new process. No dynamic loading of code or creation of instance data segments is involved. See also native system library or DLL.

TNS to native-mode access shell. A shell object file, generated by the shell generator, that supports procedure calls from TNS object files to a particular TNS/R native-mode or TNS/E native-mode library routine. The shell suspends TNS code emulation, copies and reformats parameters from the TNS execution stack to the native execution stack, calls the desired routine in native mode, copies back the function result, and resumes TNS code emulation. A custom shell exists for each native-mode library routine that can be called from TNS object files.

TNS user data segment. In a TNS process, the segment at virtual address zero. Its length is limited to 128 kilobytes. A TNS program's global variables, stack, and 16-bit heap must fit within the first 64 kilobytes. See also compiler extended-data segment.

TNS user library. A user library available to TNS processes in the Guardian environment.

TNS word. An instruction-set-defined unit of memory. A TNS word is 2 bytes (16 bits) wide, beginning on any 2-byte boundary in memory. See also Intel Itanium word, MIPS RISC word, and word.

TNS/E. Refers to fault-tolerant HP computers that support the HP NonStop operating system and are based on the Intel Itanium processor. TNS/E systems run the Itanium instruction set and can run TNS object files by interpretation or after acceleration. TNS/E systems include all HP NonStop systems that use NSE-*x* processors. Contrast with TNS and TNS/R. See also Explicitly Parallel Instruction Computing (EPIC).

TNS/E library. A TNS/E native-mode library. TNS/E libraries are always dynamic-link libraries (DLLs); there is no native shared run-time library (SRL) format.

TNS/E native C compiler. The C compiler that generates TNS/E object files. Contrast with TNS C compiler and TNS/R native C compiler.

TNS/E native compiler. A compiler in the TNS/E development environment that generates TNS/E native object code, following the TNS/E native-mode conventions for memory, stack, registers, and call linkage. The TNS/E native C compiler is an example of such a compiler. Contrast with TNS compiler and TNS/R native compiler.

TNS/E native mode. The primary execution environment on a TNS/E system, in which native-compiled Intel Itanium object code executes, following TNS/E native-mode compiler conventions for data locations, addressing, stack frames, registers, and call linkage. Contrast with TNS interpreted mode and TNS accelerated mode. See also TNS/R native mode.

TNS/E native object code. The Intel Itanium instructions that result from processing program source code with a TNS/E native compiler. TNS/E native object code executes only on TNS/E systems, not on TNS systems or TNS/R systems.

TNS/E native object file. An object file created by a TNS/E native compiler that contains Intel Itanium instructions and other information needed to construct the code spaces and the initial data for a TNS/E native process.

TNS/E native process. A process initiated by executing a TNS/E native object file. Contrast with TNS process and TNS/R native process.

TNS/E native signal. A signal model available to TNS/E native processes in both the Guardian and Open System Services (OSS) environments. TNS/E native signals are used for error exception handling.

TNS/E native user library. A user library available to TNS/E native processes in both the Guardian and Open System Services (OSS) environments. A TNS/E native user library is implemented as a TNS/E native dynamic-link library (DLL).

TNS/R. Refers to fault-tolerant HP computers that support the HP NonStop operating system and are based on 32-bit reduced instruction-set computing (RISC) technology. TNS/R systems run the MIPS-1 RISC instruction set and can run TNS object files by interpretation or after acceleration. TNS/R systems include all HP systems that use NSR-*x* processors. Contrast with TNS and TNS/E.

TNS/R library. A TNS/R native-mode library. For a PIC-compiled application, TNS/R libraries can be dynamic-link libraries (DLLs) or hybridized native shared runtime libraries (SRLs). For an application that is not PIC compiled, TNS/R libraries can only be native SRLs.

TNS/R native C compiler. The C compiler that generates TNS/R object files. Contrast with TNS C compiler and TNS/E native C compiler.

TNS/R native compiler. A compiler in the TNS/R development environment that generates TNS/R native object code, following the TNS/R native-mode conventions for memory, stack, 32-bit registers, and call linkage. The TNS/R native C compiler is an example of such a compiler. Contrast with TNS compiler and TNS/E native compiler.

TNS/R native mode. The primary execution environment on a TNS/R system, in which native-compiled MIPS object code executes, following TNS/R native-mode compiler conventions for data locations, addressing, stack frames, registers, and call linkage. Contrast with TNS interpreted mode and TNS accelerated mode. See also TNS/E native mode.

TNS/R native object code. The MIPS RISC instructions that result from processing program source code with a TNS/R native compiler. TNS/R native object code executes only on TNS/R systems, not on TNS systems or TNS/E systems.

TNS/R native object file. An object file created by a TNS/R native compiler that contains MIPS RISC instructions and other information needed to construct the code spaces and the initial data for a TNS/R native process.

TNS/R native process. A process initiated by executing a TNS/R native object file. Contrast with TNS process and TNS/E native process.

TNS/R native shared run-time library (TNS/R native SRL). A shared run-time library (SRL) available to TNS/R native processes in both the Guardian and Open System Services (OSS) environments. TNS/R native SRLs can be either public or private. A TNS/R native process can have multiple public SRLs but only one private SRL.

TNS/R native signal. A signal model available to TNS/R native processes in both the Guardian and Open System Services (OSS) environments. TNS/R native signals are used for error exception handling.

TNS/R native user library. A user library available to TNS/R native processes in both the Guardian and Open System Services (OSS) environments. A TNS/R native user library is implemented as a special private TNS/R native shared run-time library (TNS/R native SRL).

TNSVU. A tool used on TNS/E systems to browse through TNS object files that have been accelerated by the TNS Object Code Accelerator (OCA). TNSVU displays Intel Itanium code in addition to TNS code.

word. An instruction-set-defined unit of memory that corresponds to the width of registers and to the most common and efficient size of memory operations. A TNS word is 2 bytes (16 bits) wide, beginning on any 2-byte boundary in memory. A MIPS RISC word is 4 bytes (32 bits) wide, beginning on any 4-byte boundary in memory. An Intel Itanium word is also 4 bytes (32 bits) wide, beginning on any 4-byte boundary in memory.

$RECEIVE. A special file name through which a process receives and optionally replies to messages from other processes.

Index