Native Inspect Manual

HP Part Number: 528122-014
Published: February 2012
Edition: J06.12 and subsequent J-series RVUs and H06.23 and subsequent H-series RVUs

Contents

About This Document

This manual describes the use of the Native Inspect symbolic command-line debugger for HP NonStop TNS/E systems.

Supported Release Version Updates (RVUs)

This manual supports J06.03 and all subsequent J-series RVUs and H06.13 and all subsequent H-series RVUs until otherwise indicated in a replacement publication.

Intended Audience

This manual is intended for anyone who wants to debug TNS/E native processes or snapshot files using a command-line debugger on a TNS/E system.

New and Changed Information

New and Changed Information for H06.24/J06.13 (528122-014)

The H06.24 and J06.13 version of the manual contains the following enhancements:

- Updated the Origins of Native Inspect (page 16) section.
- Added a new compression option for the save Command (page 109).
- Added a new library yrtcdll for 64-bit to the Debugging Memory Problems (page 29) section, the info Command (memory leak detection) (page 90) and the set heap-check Command (memory leak detection) (page 115).
- Added a new note to the Tcl Commands Provided by Native Inspect (page 133) section and to Table 14 (page 133).

New and Changed Information for H06.23/J06.12 (528122-013)

The H06.23 and J06.12 version of the manual contains the following enhancements:

- Added a new note, NOTE (page 90).
- Added a sentence as a fourth note,NOTE (page 103).

New and Changed Information H06.21/J06.10 (528122-012)

The H06.21 and J06.10 version of the manual contains the following enhancements:

- Added a new section, Displaying the Length of the COBOL Variables (page 50).
- Updated the syntax and the description of attach Command (page 66).
- Updated the syntax and the description of detach Command (page 73).
- Updated the syntax and the description of vector Command (page 124).

New and Changed Information for H06.20/J06.09 (528122-011)

The H06.20 and J06.09 version of the manual contains the following enhancements:

- Added examples in the Examples (page 68) section of the break (tbreak) Command (page 67).
- Updated the description for the following attributes of map-source-name (map):
  - *`source-name`*
  - *`source-prefix`*
- Updated the description of what column header of info Command (page 86).

New and Changed Information for H06.20/J06.09 and H06.17/J06.06 (528122–009)

The H06.16 and J06.05 version of the manual contains the following enhancements:

- Added these commands and their descriptions to Utility Commands in Table 11: Native Inspect Command Functions (page 56), and added the commands, syntax, and examples to the commands section:
  - `define` *`commandname`*
  - `document` *`commandname`*
  - `show user [`*`commandname`*`]`
- Added `set optimized-loc-print` *`n`* and an example of its usage to set Command (environment) (page 110).

Miscellaneous changes include:

- Changed the wording under nocstm Option (page 103) to refer to the custom startup file, described under Reading the Custom File (page 23).
- Clarified text under Debugging TNS Processes (page 27).

New and Changed Information for H06.14/J06.03 and H06.15/J06.04 (528122–008)

New and Changed Information for H06.14/J06.03 and H06.15/J06.04 (528122–008)

The H06.14, J06.03, H06.15, and J06.04 version of the manual contains the following enhancements:

- Added the new fopen Command (page 81), with its description, syntax, and examples. Added its summary information to Utility Commands in Table 11 (page 56).
- Added information on the new functionality of the print Command (page 103) that enables you to invoke command line function calls in the program being debugged from the debugger command line.
- Added a new section, Debugging Memory Problems (page 29), to indicate how the Native Inspect debugger can find memory leaks, view heap usage, and detect related problems.
- Added new memory leak detection commands and options to existing commands to Chapter 4: Native Inspect Command Syntax. These commands and options are summarized under Categories of Native Inspect Commands (page 56) and at the end of Table 11 (page 56). The new commands are:
  - info Command (memory leak detection) (page 90)
  - set heap-check Command (memory leak detection) (page 115)

  The heap-check option is added to the show Command (page 116).
- Added the `optimized-code-warning [off |on ]` option to the set Command (environment) (page 110). This option's status can be displayed using the show Command (page 116).

Miscellaneous changes include:

- Under set Command (environment) (page 110), changed the mode attribute to `mode {user | priv [on | off]}`.
- Added section, Stopping Mid-Statement (page 40), to indicate how Native Inspect handles situations when the debugger is not stopped at the starting instruction of a source statement.
- Removed the glossary and put the terms and definitions in the *NonStop System Glossary*.

- Under the save Command (page 109), indicated that snapshot files of file code 130 are used for offline debugging.
- Added the process entered debug event to Table 6 (page 31).
- Updated Document Organization (page 11) to include Appendix B.
- Changed all references to *`linespec`* to *`locspec`*. See Syntax of locspec (page 59), which also indicates that *`locspec`* is sometimes referred to as *`linespec`* in files and documents that are used by or related to Native Inspect.

Made miscellaneous formatting corrections to improve accuracy and consistency.

Document Organization

This document is organized as follows:

Table 1 Contents of the *Native Inspect Manual*

Chapter	Description
Chapter 1: Introducing Native Inspect	Describes the basic principles of using Native Inspect to debug TNS/E native processes and snapshot files.
Chapter 2: Using Native Inspect	Presents a sample session using Native Inspect to debug a TNS/E native process.
Chapter 3: Using Native Inspect With COBOL Programs	Describes features and behavior of Native Inspect that are specific to the debugging of COBOL programs.
Chapter 4: Native Inspect Command Syntax	Lists Native Inspect commands in related groupings and gives syntax for all the Native Inspect commands and command-line options.
Chapter 5: Using Tcl Scripting	Describes the commands of the Tool Command Language (Tcl)
Appendix A: Command Mapping With Debug and Inspect	Lists Debug and Inspect commands with their Native Inspect equivalents.
Appendix B: Redirected and Aliased WDB Debugger Commands	Shows the correspondence between WDB and Native Inspect commands.

Notation Conventions

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

`Italic Letters`

Italic letters, regardless of font, indicate variable items that you supply. Items not enclosed in brackets are required. For example:

`file-name`

`Computer Type`

Computer type letters indicate:

- C and Open System Services (OSS) keywords, commands, and reserved words. Type these items exactly as shown. Items not enclosed in brackets are required. For example:

  Use the `cextdecs.h` header file.

- Text displayed by the computer. For example:

```
Last Logon: 14 May 2006, 08:02:23
```

- A listing of computer code. For example:

  ```
  if (listen(sock, 1) < 0)
  {
  perror("Listen Error");
  exit(-1);
  }
  ```

Bold Text

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

```
ENTER RUN CODE

?123
CODE RECEIVED:      123.00
```

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

[ ] 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 ) ;
```

In the following example, space is permitted (but not required) following the comma. Space is not permitted following the period :

```
STATUS *, TERM $ZTN2.#PT4UCAF
```

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]…
```

Related Information

Three debuggers are available for use when debugging on TNS/E systems, as described in Table 2.

Table 2 Related Debugging Tools and Documentation

Target Process	Supported Debugger	User Information
TNS/E native processes on a TNS/E system.	Native Inspect, a command-line debugger running on the TNS/E host.	*Native Inspect Manual* and Native Inspect online help.
TNS/E native processes on a TNS/E system.	Visual Inspect, a GUI debugger running on a Microsoft Windows Workstation connected to the TNS/E host.	Visual Inspect online help.
Emulated TNS processes on a TNS/E system (either accelerated by OCA or interpreted by OCI).		
Emulated TNS processes on a TNS/E system (either accelerated by OCA or interpreted by OCI).	Inspect, a command-line debugger running on the TNS/E host.	*Inspect Manual*.

Publishing History

The publishing history for this volume is shown in Table 3 (page 13).

Table 3 Part Numbers and Publication Dates

Part Number	Product Version	Publication Date
528122–003	Native Inspect H01	July 2005
528122–004	Native Inspect H01	November 2005
528122–005	Native Inspect H01	May 2006
528122–006	Native Inspect H01	August 2006
528122–007	Native Inspect H01	February 2008
528122–008	Native Inspect H01	August 2008

Table 3 Part Numbers and Publication Dates *(continued)*

Part Number	Product Version	Publication Date
528122–009	Native Inspect H01	February 2009
528122–011	Native Inspect H01	February 2010
528122-012	Native Inspect H01	August 2010
528122-013	Native Inspect H01	September 2011
528122-014	Native Inspect H01	February 2012

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 Introducing Native Inspect

Native Inspect on TNS/E Systems

Native Inspect is a system-level command-line symbolic debugger that you use to debug TNS/E native processes and snapshot files. Native Inspect runs only on a NonStop TNS/E host system (not on a TNS/R or TNS system).

You can use Native Inspect to do the following tasks:

- To debug TNS/E native programs or snapshot files written in TNS/E native C/C++, pTAL, or TNS/E native COBOL, especially when Visual Inspect is not available
- To use a Tool Command Language (Tcl) script for automating debugging operations on TNS/E native debugging targets
- To perform privileged or global debugging on TNS/E native programs

Debuggers on NonStop TNS/E Systems

TNS/R systems support two command-line debuggers: Debug and Inspect. On a TNS/E system, Native Inspect replaces both Debug and Inspect. Debug does not exist on TNS/E systems. Inspect is present on TNS/E systems, but you can use it only to debug **emulated** TNS processes. You cannot use Inspect to debug native TNS/E processes and snapshot files. Table 4 summarizes the debugger availability by system type.

Table 4 Debugger Availability by System Type

System	Debug	Inspect	Native Inspect	Visual Inspect
TNS/R	Yes	Yes	No	No
TNS/E	No	Emulated	Yes	Yes[1]

[1] Requires a Windows workstation.

Table 5 provides usage scenarios for Native Inspect, Visual Inspect, and Inspect on a TNS/E system. For each debugger, symbolic debugging is available for code optimized at level 0 or 1, and debugging support is limited for code optimized at level 2.

Table 5 Using Debugging Options on TNS/E Systems

Debugger to Use	ID	Type or State You Can Debug	Languages Supported
Native Inspect on NonStop TNS/E systems.	T1237	TNS/E native processes (executing in the same CPU).	TNS/E native C/C++
		Snapshot files that save the state of TNS/E native processes.	TNS/E native COBOL
			pTAL
Visual Inspect on Windows.[1]	T7877, T9756, T9226, T9673.	TNS/E native processes.	TNS/E native C/C++
		TNS/E and TNS/R native snapshot files.	TNS/E native COBOL
		Emulated TNS processes.	pTAL
Inspect on NonStop TNS/E.	T9673	Emulated TNS processes on TNS/E systems (either accelerated with OCA or interpreted with OCI).	TNS Fortran
		Snapshot files of all processes (TNS/R native, TNS/E native and nonnative).	TNS COBOL
			TAL
			TNS C/C++
			Screen COBOL

[1] This is preferred debugger on TNS/E systems.

Origins of Native Inspect

The first release of the Native Inspect application was based on the Free Software Foundation GDB (Gnu) debugger (version 4.16). In H06.13, the Native Inspect application was derived from WDB (Wildebeest) command-line debugger (version 5.5) and from Tool Command Language (Tcl) version 8.0.4. This release of the Native Inspect application is derived from GDB version 6.8 and Tcl version 8.5. It continues to have the features that were leveraged from WDB version 5.5. Information about redirected and aliased WDB commands is provided in Appendix B (page 139).

GDB Industry Standard, Open Source Debugger

The GDB debugger supports many platforms and is an industry-standard debugger. Like both GDB and WDB, Native Inspect is open source, and much of the functionality of Native Inspect is the same as that of GDB and WDB.

If you are familiar with GDB or WDB, you should find that Native Inspect is similar in many ways. Documentation available for GDB or WDB might yield information that also applies to Native Inspect.

Additional NonStop Extensions

Native Inspect includes many functions specific to NonStop systems, such as support for HP NonStop operating system extensions to the symbol table format, multiprocess debugging support, COBOL support, and support for EDIT files and memory access breakpoints or MABs.

Relationship to the Inspect Subsystem

Native Inspect originated in the UNIX environment and is therefore not part of the Inspect subsystem. However, both Inspect and Visual Inspect are part of the Inspect subsystem.

Documentation for Native Inspect

The following documentation is relevant when using Native Inspect:

- Native Inspect-Specific:
  - *Native Inspect Manual* (this manual).
  - *Native Inspect Quick Reference Card.*
  - Native Inspect online help, using the `help` command, or the `help` option. (See the help Command, help Option (page 83).)
- Related Documents:
  - Hewlett-Packard home page for WDB:

    http://www.hp.com/go/WDB
  - The user manual for GDB, titled *Debugging with GDB* by Richard M. Stallman and Roland H. Pesch. This document is freely available in various formats on the World-Wide Web. Use your preferred search engine to locate a copy by searching on the title.
  - Current GDB documentation:

    http://www.gnu.org/software/gdb/

Comparing Native Inspect to Debug

A summary of differences is provided in the following table:

Native Inspect	Debug
The built-in debugger on TNS/E systems.	Debugger of last resort on TNS/R and TNS systems; not available on TNS/E systems.
A separate product.	Part of the HP NonStop operating system.
Executes as a separate process from the process being debugged.	Executes in the context of the process being debugged.

Native Inspect takes the place of Debug as the low-level default process debugger on TNS/E systems. Therefore, Native Inspect is the debugger invoked if Visual Inspect (the preferred debugger on TNS/E systems) is not available (for example, if no connection exists to a Windows client) or if the INSPECT parameter is set to OFF.

Debugger selection criteria are shown in Figure 2 (page 21) and in Figure 3 (page 22).

Debug was an integral part of the operating system on previous NonStop platforms. On TNS/E, Native Inspect is a separate licensed object file (`$SYSTEM.SYSnn.EINSPECT`), but still fulfills the role of built-in debugger. Whereas Debug executed in the context of the process being debugged, Native Inspect executes as a separate process in the same CPU as the process being debugged. Running as a separate process reduces the chances of the debugger affecting target process behavior.

Some Commands Are Debug-Compatible

Although Native Inspect is completely different in form from Debug, Native Inspect provides many Debug-compatible commands, such as `a`, `base`, `fn`, `ih`, and `mh`. For a list of Debug commands that have equivalent Native Inspect commands, see Appendix A: Command Mapping With Debug and Inspect.

Comparing Native Inspect to Inspect

Native Inspect and Inspect are both command-line debuggers, but there are many differences in the command sets of the two debuggers. Native Inspect commands and output formats are based on GDB and are therefore very different from the commands and output formats of Inspect. Inspect users should approach Native Inspect as a new product equipped with the additional power of Tcl scripting.

Differences Between Native Inspect and Inspect

- Command names for Inspect and Native Inspect are different. For a comparison, see the following:
  - Table 16 (page 136) for a list of Inspect commands and equivalent Native Inspect commands.
  - Table 8 (page 33) for a list of the most commonly used Native Inspect commands with their Inspect equivalents.
- The Inspect `STEP OVER`, `STEP IN`, and `STEP` functions are provided by the `nexti` command, `stepi` command, and `until` command, respectively.
- Inspect locations are scope-based (that is, function/procedure), while Native Inspect's locations are based on source file line numbers. Native Inspect, unlike Inspect, does not prefix line numbers and function names with a hash symbol (#).
- Native Inspect automatically displays the current line of source.
- The `source` command performs the same function as Inspect's `OBEY` command.

- Native Inspect does not contain formatting support for SPI buffers. You must use Visual Inspect to display SPI buffers.
- The Inspect `DISPLAY` command is used in Inspect to print variable values. In Native Inspect, the `display` command is defined to add variables and expressions to the automatic display list – a list that is automatically displayed each time the program is suspended. The Native Inspect `print` command and `output` command are equivalents to the Inspect DISPLAY function.

COBOL-Specific Differences

The following capabilities of Inspect are not supported in Native Inspect:

- Setting breakpoints using the *`program-unit.label`* or *`program-unit.line-number`* notation.
- Using PICTURE and FORMAT clauses for displaying items.
- Formatting records as an arbitrary type (`DISPLAY AS` command).
- Setting breakpoints on statement ordinals.

Process Debugging Using Native Inspect

Use Native Inspect to debug TNS/E native processes (and snapshot files) in either the Guardian or HP NonStop Open System Services (OSS) environment.

The Native Inspect licensed object file is `$SYSTEM.SYSnn.EINSPECT`. Native Inspect runs as a native, non-priv, high-pin process, separate from the process being debugged (see Comparing Native Inspect to Debug (page 17)).

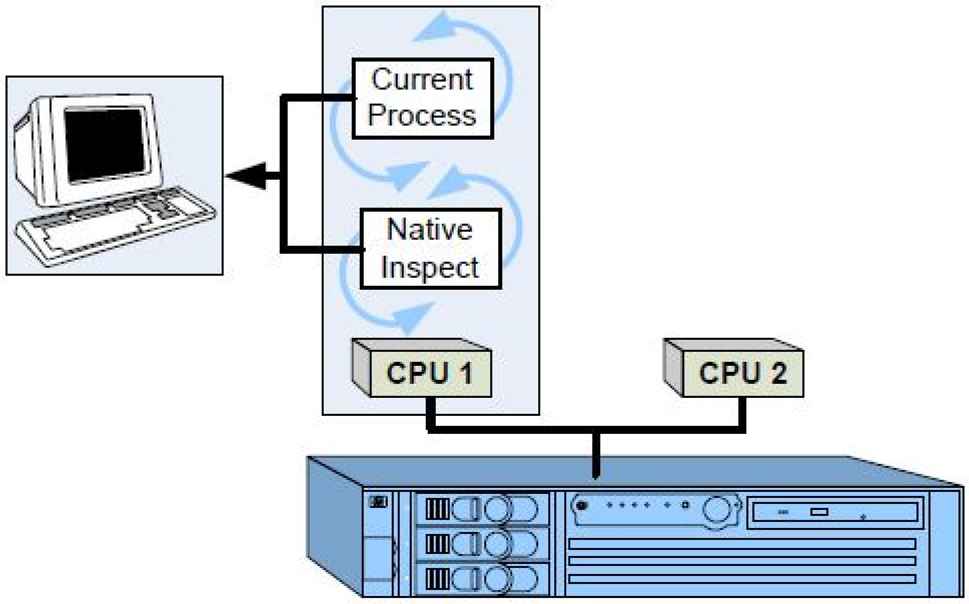
Native Inspect must always be running in the same CPU as the process you want to debug, as shown in Figure 1 (page 19).

Languages Supported by Native Inspect

- TNS/E native C/C++
- pTAL
- TNS/E native COBOL

To debug code written in any other language for TNS/E systems or snapshot files created on a TNS/R system, you must use Visual Inspect or Inspect.

Figure 1 Native Inspect Runs in Same CPU as Current Process



Starting Native Inspect

You can start or enter Native Inspect in several ways, as follows:

- You can start a process under the control of the debugger, subject to the Debugger Selection Criteria (using the TACL `RUND` command or the `run -debug` command in OSS).
- You can debug a running process (using the TACL `DEBUG` command).
- A running process can invoke the debugger (calling the `PROCESS_DEBUG_` or `DEBUG` procedure, by encountering a breakpoint, or as a result of an unhandled signal).
- You can explicitly start Native Inspect from a TACL prompt or from OSS.

These methods are described next.

Starting a Process Under the Control of the Debugger

At the TACL prompt, enter a RUND command, specifying the file name of the TNS/E native object file you want to debug as follows:

```
TACL> rund $DISK2.MYSUBVOL.MYFILE
```

In OSS, enter a `run` command with the `-debug` option and specify the file name of the TNS/E native object file you want to debug as follows:

```
OSS> run -debug usr/bin/myfile
```

Native Inspect is automatically run in the same CPU as the process ($DISK2.MYSUBVOL.MYFILE in the first example) providing that the following conditions are true:

- The target `usr/bin/myfile` is a TNS/E native program (file code 800).
- You have not set up a Visual Inspect connection to the host.

For more information about when a particular debugger is invoked, see Debugger Selection Criteria (page 20).

Debugging a Running Process

At the TACL prompt, enter a `DEBUG` command and specify the name of the TNS/E native process as follows:

```
TACL> debug $myproc
```

This command starts Native Inspect on the home terminal of the process `$myproc`.

You can optionally specify a home terminal on which you want Native Inspect to run, as follows:

```
TACL> debug $myproc, term $ztn10.#pthef
```

In OSS, enter a `DEBUG` command that includes the CPU and process numbers of the process you want to debug as follows:

```
OSS> debug 5,135, term $myterm
```

Native Inspect gains control of the running process subject to the NonStop debugging rules (described in Debugger Selection Criteria (page 20)).

For example, if a process is executing privileged code, the process must return to nonprivileged code before a nonprivileged debug request completes. If you are logged on as the super ID (255,255), you can enter the `DEBUGNOW` command, which immediately gives you access to the specified process, even if it is running privileged code.

Invoking the Debugger From a Running Process

Native Inspect is automatically started by the NonStop operating system when a debugger is required for any of the following reasons:

- TNS/E native process code calls the `PROCESS_DEBUG_` or `DEBUG` procedure.
- A TNS/E native process encounters a breakpoint set by a prior debugging session.
- The user of another debugger switches to Native Inspect.

When a running process invokes a debugger, the operating system automatically selects a debugger according to the debugger options you set (INSPECT ON or OFF) in addition to the process type (TNS versus TNS/E) and the availability of a connection to Visual Inspect, which is the preferred debugger on TNS/E systems.

Debugger Selection Criteria

The following two figures illustrate the criteria that are evaluated during debugger selection:

- Figure 2 (page 21)
- Figure 3 (page 22)

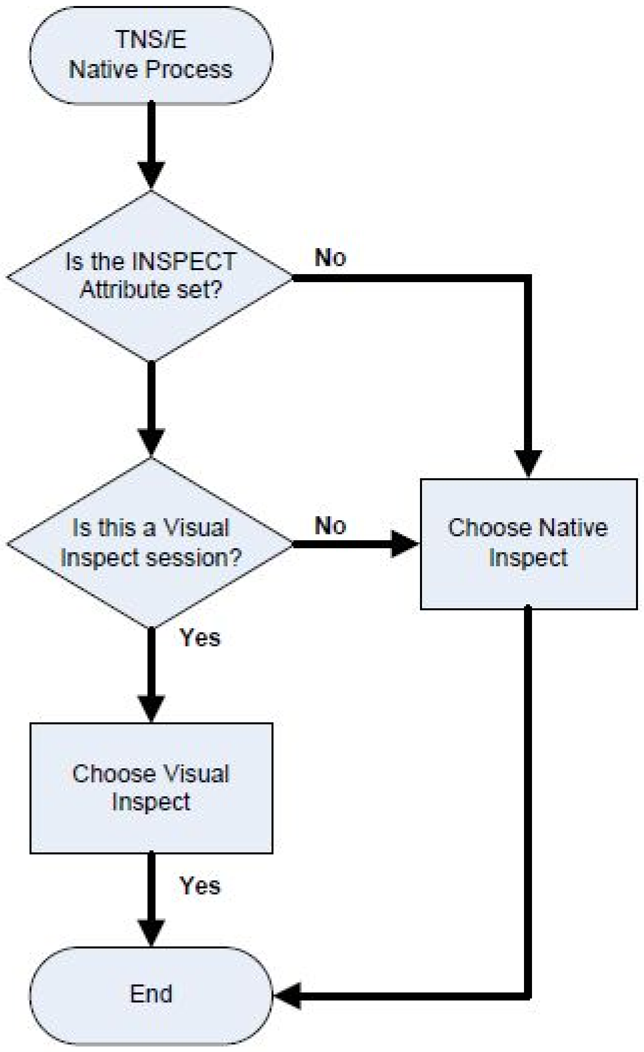
In both Figure 2 and Figure 3, debugger selection criteria are defined as follows:

Is the INSPECT attribute on?	INSPECT is set to ON for the process you will debug (set with TACL, the linker, or the `RUN[D]` command).
Is this a Visual Inspect session?	You have started Visual Inspect and have connected to the TNS/E 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.
Is Inspect available?	The Inspect subsystem (IMON, DMON, $DMnn) is running, and the Inspect command-line interface is available.

To summarize, Native Inspect is selected as the debugger under the following conditions:

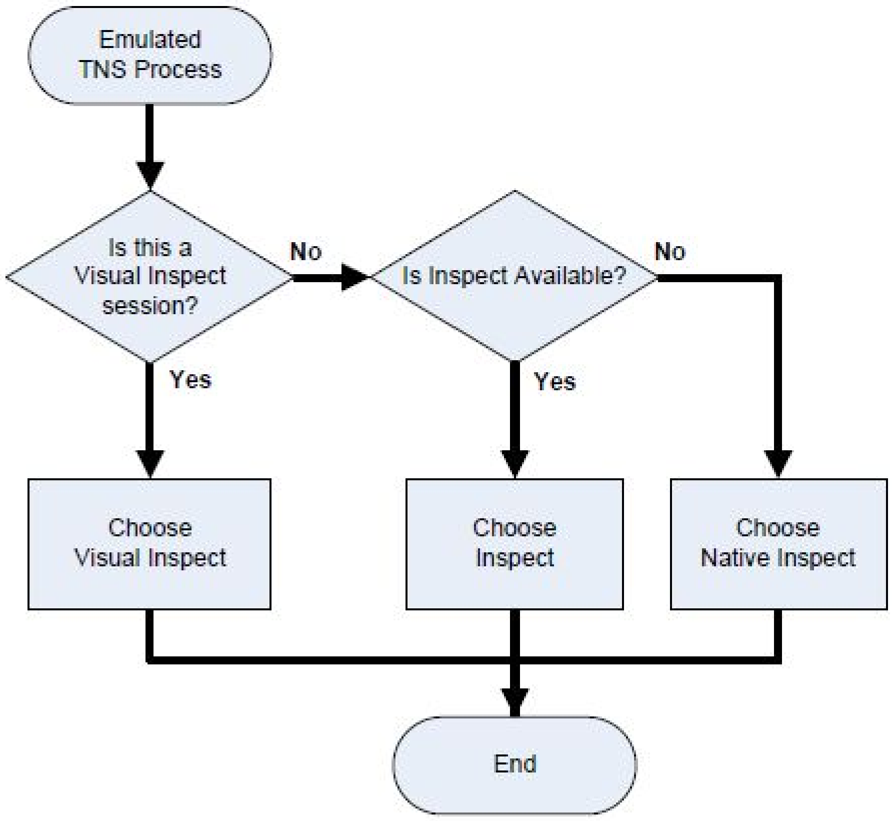
- For TNS/E native processes when you have not established a Visual Inspect connection to the NonStop host, or the INSPECT attribute is OFF (Figure 2).
- For TNS processes when the Inspect subsystem is not running (that is, neither Visual Inspect nor Inspect is available) (Figure 3). Note, however, that Native Inspect has only limited capabilities for debugging TNS processes. See Debugging TNS Processes (page 27) for more information.

Figure 2 Debugger Selection for a TNS/E Native Process



Note that in Figure 3 there is no checking for the INSPECT setting (ON or OFF) for the process. All TNS processes are given to the Inspect subsystem for debugging, so the INSPECT attribute has no effect.

Figure 3 Debugger Selection for a TNS Process Running on TNS/E



If you are using Native Inspect and the current process is a TNS process on a TNS/E system, the capabilities available to you are described in Debugging TNS Processes (page 27).

Explicitly Starting Native Inspect

At the TACL prompt, invoke the Native Inspect object file (EINSPECT) explicitly, using the `RUN` command, or implicitly, by entering the filename alone. Use the CPU option to select the CPU in which the process you wish to debug is running.

For example, use the following procedure to start Native Inspect in CPU 2 in the Guardian environment:

1. At the TACL prompt, invoke EINSPECT, as follows:

   ```
   \SYSTEM.$D0117.INSPECT 1> einspect / cpu 2 /
   ```

2. The Native Inspect start up screen is displayed as follows:

   ```
   TNS/E eInspect gdb Debugger [T1237 - 20-Dec-2011 16:43]
   Copyright 2008 Free Software Foundation, Inc.
   Copyright 2003-2012 Hewlett-Packard Development Company, L.P.

   Native Inspect (based on GDB) is covered by the GNU General Public License.
   Type "show copying" for conditions for changing and/or distributing copies.
   Type "show warranty" for warranty/support information.

   Working directory \PELICAN.$SYSTEM.STARTUP.
   (eInspect 2,-2):
   ```

3. Enter debugger commands at the following system prompt:

   ```
   (eInspect 2,-2): help
   ```

To start Native Inspect in CPU 2 in the OSS environment, enter:

```
/G/SYSTEM/SYSTEM gtacl -cpu 2 -p einspect
```

```
(eInspect 2,-2):
```

After Native Inspect initializes, you must enter the `attach` command so that you can examine a TNS/E native process. See the attach Command (page 66). The process must be running under your user ID (or you must be either the super ID or the group manager of the user), and must be running in the same CPU as the instance of Native Inspect you started.

To examine a snapshot file after starting Native Inspect, use the `snapshot` command. See the snapshot Command (page 117).

For more information about starting Native Inspect and about accessing source and loading symbols, in addition to an extended example of a Native Inspect session, see Chapter 2: Using Native Inspect.

Reading the Custom File

NOTE: Native Inspect does not execute commands in the custom file if you specify the "no custom" (`nocstm`) option.

When Native Inspect initializes, it reads the contents of its custom file, named `EINSCSTM`, located in your logon default `$vol.subvol` on the TNS/E host. This file can contain Native Inspect commands that you want to take effect during initialization. For example, you might establish your typical debugging environment by including a number of `set` commands. See the set Command (environment) (page 110).

Only a limited subset of Native Inspect commands are allowed to be specified in the `EINSCSTM` file. These include (but are not limited to) the following commands: `set`, `show`, `priv`, `define`, `document`. Process control and execution control commands are not allowed to be specified in the `EINSCSTM` file.

Using the Command Prompt to Identify the Current Process

After Native Inspect initializes, it displays its command prompt. The Native Inspect prompt contains the name `eInspect`, the CPU number, and the process identification number (pin) of the current process or debugging target.

For example, the following prompt indicates that process 0,301 is the current process:

```
(eInspect 0,301):
```

The current process is the process to which all debugging commands apply and the process for which Native Inspect waits for events.

When Native Inspect has no current process, the command prompt includes the CPU number and "`-2`" as the process number. For example:

```
(eInspect 3,-2):
```

Native Inspect Command Abbreviations and Command Alternates

You can truncate Native Inspect command names provided that the abbreviation is not ambiguous.

Ambiguous abbreviations are allowed. For example, the letter `s` is defined as an abbreviation for the `step` command even though many other command names begin with s. Use the help command to display command information. See help Command, help Option (page 83).

Some commands have alternate forms such as the `disassemble` command, which has the alternate form `da`. Where applicable, these alternate forms are identified in the headings of each command, such as: disassemble (da) Command (page 75).

Debugging Multiple Processes

Native Inspect provides support for debugging multiple TNS/E native processes in both the Guardian and OSS environments. Native Inspect does not provide the same level of multiprocess debugging capabilities that Inspect has traditionally provided and that is available on TNS/E systems by debugging with Visual Inspect.

Multiprocess debugging is easiest using either Visual Inspect or separate instances of Native Inspect.

If you use Native Inspect for multiprocess debugging, you can choose either of the following strategies:

- Use a separate Native Inspect instance for each process so that you are issuing debug requests from separate terminal sessions.
- Use one instance of Native Inspect for all processes so that one process is the current process and the other processes are in the background.

Debugging Two Processes With One Instance of Native Inspect

Suppose that you are running Native Inspect and are debugging a process (this is the "current process" of the debugger, or the debugging target). Another process will automatically be given to your existing instance of Native Inspect if a TNS/E native process causes a debugger to be invoked, and the TNS/E process is running in the same CPU and has the same user ID as your current process (or if you are using the super ID).

Two processes are then under the control of one instance of Native Inspect. See Example of Using Multiple Instances of Native Inspect (page 26).

Using One Instance of Native Inspect to Debug Multiple Processes

You need to be mindful of which process is the current process (as indicated by the `CPU,pin` in the eInspect prompt). With Native Inspect, most debugging options (such as the directory search path) are defined as attributes of the debugger instance, not as values associated with the current process.

Suppose that you are debugging two interacting processes. If you are stepping execution in process A, which causes a breakpoint in process B to be hit, process B becomes the current process. Advancing execution in process B might then result in completion of the step of process A, causing it once again to be the current process.

In this situation, you will see the process ID of process A (3,301) displayed in the Native Inspect prompt, and then the process ID of process B in the next prompt from Native Inspect (3,38). For example:

```
(eInspect 3,301): step
(eInspect 3,38):
```

Considerations for Multi-Process Debugging

- Native Inspect can debug only those processes that are executing in the same CPU in which the instance of Native Inspect is running.

  For example, if a TNS/E native process running in CPU 3 calls Debug, and Native Inspect is invoked as the debugger, the instance of Native Inspect must run in CPU 3.
- Native Inspect checks only for events at times when events are expected for the current process—such as after process execution commands (or the `wait` command). Native Inspect cannot receive events while prompting for user input.

  Debugging events that occur for other processes are not necessarily reported when they occur, but are reported when Native Inspect checks for events for the current process.
- Pressing the **Break** key interrupts waiting for a debugging event and redisplays the Native Inspect command prompt, but cannot be used to interrupt other commands.

- When debugging multiple programs, Native Inspect, like Inspect, sets the designated current process to be the process for which the most recent debugging event has been reported. The current process is the process to which all commands apply and for which Native Inspect waits for events.
- Use the `vector` command to explicitly change the designated current process.
- In a rare situation, two instances of Native Inspect can be running on different CPUs but be prompting the same home terminal. In this situation, you should do the following:
  1. Start another Native Inspect instance on one of the CPUs.
  2. Attach the process being debugged in that CPU to the new Native Inspect instance.
  3. Detach that process from the original Native Inspect instance.
- Use the `info` command with the `sessions` option to display information about the processes Native Inspect is currently debugging.
- You cannot use the automatic display list (expressions that are automatically displayed by the `display` command when program execution is suspended) when you are debugging multiple processes.

Identifying Additional Processes

Native Inspect informs you that it has control of another debugging target after you do one of the following:

- Enter a process-control command (which causes Native Inspect to wait for events), such as the `continue` command, `finish` command, `next` command, `nexti` command, `step` command, `stepi` command, and `until` command.
- Enter the `wait` command.
- Press the **Break** key.

In this case, when you use Native Inspect to debug multiple processes, the current process is the one for which the most recent debugging event has been reported.

Using a Separate Instance of Native Inspect for Each Process

If you have two processes under the control of one instance of Native Inspect, you can transfer one of the processes to another instance of Native Inspect as follows:

1. Start a new Native Inspect instance on a different terminal session, specifying the same CPU as the original instance of Native Inspect as follows:

   ```
   TACL> status *, term
   Process     Pri PFR  %WT Userid  Program file           Hometerm
   $Z160 3,301 123   R  000   8,12  $SYSTEM.SYS00.EINSPECT $ZP1.#PHF

   TACL> einspect /cpu 3/
   ```

   Each terminal session defines a different “home terminal”. However, you can transfer a process to a Native Inspect instance on a different terminal session as long as the new instance of Native Inspect is running in the same CPU and under the same user ID.

   Remember that while you are debugging this second process on a different terminal session, the home terminal for the process itself remains unchanged, and the process might attempt to prompt or output to its home terminal, causing more contention. Because of this potential problem, use Visual Inspect as the preferred debugger for multiprocess debugging.
2. Transfer the current process to the new Native Inspect instance as follows:
   a. From the new Native Inspect instance, enter the `attach` command.
   b. From the original Native Inspect instance, release the current process (the newer debugging target) by entering the `detach` command.

Example of Using Multiple Instances of Native Inspect

You are running Native Inspect in CPU 3 and are debugging the process `$DISK1.MYSUBVOL.MYPROG` (CPU,pin=3,301). A second debugging target named `$DISK2.SVOL2.YOURPROG` (CPU,pin=3,32) is assigned to your instance of Native Inspect.

After a debugging event is reported for the new current process, the following command prompt indicates that the current process has changed:

```
(eInspect 3,301):
   < debugging event is reported here >
(eInspect 3,32):
```

To have a separate Native Inspect instance for each process being debugged, you need to attach the new process (3,32) to a new Native Inspect instance, and then detach the process from the original Native Inspect instance as follows:

1. Start a second Native Inspect instance in a different terminal session but running on the same CPU:

   ```
   TACL > einspect / cpu 3/
   (eInspect 3,-2):
   ```

2. Using this new Native Inspect instance, enter an `attach` command, specifying the process ID of the process you want to transfer as follows:

   ```
   (eInspect 3,-2): attach 32
   ```

   The `attach` command does not complete until you perform the final step (`detach`).

3. Using the original Native Inspect instance, enter the `detach` command as follows:

   ```
   (eInspect 3,32): detach
   ```

   The preceding `detach` command releases the new process to the attaching Native Inspect instance and allows the original process back to its original instance of Native Inspect.

4. The new terminal session indicates that the attach has completed, and process `3,32` is the current process for the new instance of Native Inspect:

   ```
   (eInspect 3,-2): attach 32

   Symbols read in for program loadfile
   \PIPPIN.$D0117.TESTS.ODISP3.

   Process (3,32) received DS_EVENT_ENTER_DEBUG
   (eInspect 3,32):
   ```

You now have two instances of Native Inspect attached to each of the two programs being debugged in CPU 3.

Global Debugging

A privileged user (the super ID user) can set one or more global breakpoints (breakpoints set in code that is shared by multiple processes, such as DLLs). Any process that encounters the global breakpoint comes under the control of Native Inspect. Such a debug session is defined as global debugging.

Native Inspect Is the Global Debugger

On the TNS/E system, only Native Inspect can do global debugging, and only one global debugging session can exist in a CPU at any one time. The Native Inspect instance that sets a global breakpoint is implicitly registered as the global debugger in that CPU when the first global breakpoint is set, and the Native Inspect instance is deregistered when the last global breakpoint is removed.

Global debugging is privileged debugging and is also a special case of multiprocess debugging (see Debugging Multiple Processes (page 24)). The super ID user must use one instance of Native

Inspect (that is, the registered global debugger) to debug all processes in a CPU that encounter the global breakpoints.

Considerations for Global Debugging

- Privileged mode must be enabled before you can set breakpoints in or examine privileged code. You must logon with the super ID and explicitly enable privileged mode debugging by entering the `priv` command.
- To set a global breakpoint, you must enable privileged debugging and then specify the `-g` flag when setting a breakpoint. Global breakpoints are triggered by any process that executes the code on which the breakpoint is set.
- Global debugging mode begins when the first global breakpoint is set, and ends when the last global breakpoint is deleted. While global debugging is in effect, all other debugging in the CPU is suspended.
- Native Inspect detects debugging events only when it is waiting. The debugger cannot detect debugging events when prompting the user for input. For this reason, you should periodically issue the `wait` command so that Native Inspect can detect any debugging events that might have occurred. When waiting, you can press the **Break** key to redisplay the command prompt.
- The super ID has the unique ability to vector to and examine processes running in the current CPU without establishing a debugging session with that process.

△ **CAUTION:** Use the super ID's `vector` capability with care. The process might be executing, and you cannot use execution control commands or commands that alter the process state.

- When privileged debugging mode is enabled, the `attach` command issues a `DEBUGNOW` request instead of a `DEBUG` request.

Debugging TNS Processes

Native Inspect does not support debugging TNS processes nor TNS snapshot files. However, a TNS process may come under the control of Native Inspect when the Inspect subsystem (T9226) is not running.

If a TNS process becomes the current process in Native Inspect, you can do the following:

- Create a snapshot of the TNS process for later analysis with Visual Inspect or Inspect (using the `save` command)
- Display a stack trace of the TNS process (using the `bt` command)
- Continue execution (using the `continue` command)
- Transfer the TNS process to Inspect once the Inspect subsystem is started (using the `switch` command)
- Stop the TNS process or exit the debugger (using the `kill` command)

For TNS programs that are executing OCA-generated TNS/E code, Native Inspect can debug the program, nonsymbolically, at the TNS/E machine level. You can use commands such as `continue`, finish, `next`, `step`, and `until`. Native Inspect applies the commands to the underlying TNS/E native view rather than to the TNS process itself. Also, the `bt` command can display a TNS stack trace.

Debugging Snapshot Files

A process snapshot file or snapshot is a disk file that is an image of a process, its data, and its status at the moment it was saved. Snapshot files are analogous to the *core* files on UNIX systems. Snapshot files have file code 130. You can use all three debuggers (Native Inspect, Visual Inspect, and Inspect) to debug snapshot files (Inspect refers to snapshot files as save files).

Creating a Snapshot File

You can create snapshot files in several ways, as follows:

- Using the `save` command (Native Inspect)
- Using the **Save Snapshot** command (Visual Inspect)
- Using the `SAVE` command (Inspect)

Snapshot files are also created by the snapshot server (`INSPSNAP`) if the `SAVEABEND` attribute for a process is `ON` and the process abends.

Opening a Snapshot file

Use the Native Inspect `snapshot` command to open a snapshot file, providing a read-only view of the state of the process that the snapshot represents.

To open a snapshot file, start Native Inspect and then enter the following command:

```
(eInspect 3,301): snapshot $disk3.mysubvol.myprog
```

where `$disk3.mysubvol.myprog` is the name of a TNS/E native snapshot file (file code 130) located on the TNS/E system.

Snapshot File Considerations

- You can examine only one snapshot file at a time, and you cannot debug a process at the same time. You cannot execute any Native Inspect commands that would alter the state of the snapshot, or commands that would execute a process, such as step or continue. Such commands report an error during snapshot analysis.
- If the snapshot was created in a different location from its present location, you might need to manually load symbols for loadfiles (using the `symbol` command or `symbol-file` command,) and set a source search path (using the `dir` command).

Debugging DLLs

Dynamic-link libraries (DLLs), which contain position-independent code (PIC), are the standard user libraries on TNS/E systems. DLLs can be implicitly loaded by the system when a program is started or explicitly loaded and unloaded by program calls to `dlopen()` and `dlclose()`, respectively.

Suspending Process Execution on DLL Events

You can use the `LOAD` and `UNLOAD` options with the `catch` command to gain control when a DLL is loaded or unloaded, respectively.

Listing DLLs

Native Inspect maintains a list of the loadfiles that compose the current program. You can display the list of loadfiles by entering the `info` command with the `dll` option.

Loading Symbols for DLLs

Native Inspect automatically loads symbols for the program file and for explicitly loaded DLLs. You must explicitly load symbols for other DLLs using the `symbol` command or `symbol-file` command. To explicitly add additional symbol files at a specific address, use the `add-symbol-file` command.

Addressing Symbols for DLLs Loaded at Another Address

If a user loads a symbol file for a DLL before a program call to `dlopen()` to load the DLL, Native Inspect uses the preferred address of the DLL as the basis for symbol address calculations. Then, when the DLL is loaded, Native Inspect again attempts to load the symbols using the actual load address for the DLL.

Setting Breakpoints

Native Inspect does not support the ability to set breakpoints on DLL functions before the DLL is loaded; you must wait until the DLL is loaded. Use the `add-symbol-file` command if you want to specify the base address relative to which symbolic addresses are calculated.

Debugging Memory Problems

You can use the Native Inspect debugger to find memory leaks, view heap usage, and detect related problems.

The following memory-related errors can occur in an application:

- Heap corruption
- Memory leaks
- Access errors

The memory leak detection functionality is only available when using the memory leak detection special library, ZRTCDLL (in 32-bit)/YRTCDLL (in 64-bit), found in the $SYSTEM.SYS*nn* subvolume. The library must be specified as an "interpose library" when the application is run using the `lib` option.

For example, if the application does not specify the ZRTCDLL library, it will get the following message:

```
(eInspect 1,598):set heap-check on
ZRTCDLL library not found. To use runtime checking this library
must be specified using the "lib" option.
```

To avoid this message, the application must be run with the `lib` option. For example:

For 32-bit application:

```
rund application /lib $system.sys00.zrtcdll/
```

For 64-bit application:

```
run –debug –lib=/G/system/sys00/yrtcdll application
```

NOTE: If your application already uses an interpose library, it is not possible to add the memory leak detection library and thus obtain memory leak detection functionality.

Heap Corruption

A heap corruption occurs when an application erroneously overwrites some of the data in the heap. Heap corruption can result in data corruption, memory corruption, or both. When an application inadvertently uses the erroneously overwritten data in the heap, it results in data corruption in the application. Data corruption can lead to unpredictable program behavior. The data corruption in the heap can lead to memory corruption if the corrupted data in the heap is used by memory management functions in the application to allocate, access, or deallocate memory blocks.

In other words, memory corruption occurs when the corrupted datum in the heap is accessed as a pointer. Memory corruptions compromise the data integrity of the application and can result in segmentation violations if the erroneously allocated or accessed memory blocks are out of the bounds of the virtual memory of the application.

Memory Leaks

A memory leak occurs when an application fails to free allocated memory. As a result, the kernel frees the memory that is allocated by a process only when the process terminates. If the program leaks memory on a continual basis, the virtual memory requirement for the process continues to increase and this can result in serious consequences for long-running applications and memory intensive applications.

Memory leaks can also cause fragmentation of the heap. This slows down the allocation, de-allocation, and access of memory blocks and can eventually cause the application to fail with out-of-memory errors.

You should suspect a memory leak in an application if the system runs out of swap space, runs slower, or both. Memory leaks in an application increase the memory consumption in an application. When the memory consumed by the application exceeds the resource limits set by the kernel, the application fails with out-of-memory errors.

NOTE: For programs compiled at all optimization levels, a memory leak may be reported at a line in the code that does not match the actual leak location. This only happens when the memory allocation and the leak happen in the same block of statements containing no control flow (which is not common).

Access Errors

Memory access errors can occur under the following conditions:

- When reading uninitialized local, or heap data
- When reading or writing to nonexistent or unallocated memory
- When a stray pointer overflows the bounds of a heap block, or tries to access a heap block that is already freed to cause buffer overruns and underruns
- When reading or writing to memory locations that are already freed in the program

Commands For Interactive Memory Debugging

You can use the memory leak detection commands, described in Chapter 4: Native Inspect Command Syntax, to debug memory problems through Native Inspect.

For more information on memory debugging with Native Inspect, see the *Debugging Dynamic Memory Usage Errors Using HP WDB* white paper and the *Debugging with GDB* manual at the HP WDB Documentation webpage: http://www.hp.com/go/WDB. Native Inspect's implementation of memory debugging is similar to that of WDB.

Handling Events

When Native Inspect receives an event, it informs you what event has occurred and then updates the debugging context (that is, information specific to the current process, such as breakpoints, loaded DLLs, and the current register values).

The type of events that have the most impact on debugging with Native Inspect are debugging events, which typically suspend a program under debugger control. For example, a breakpoint is a debugger-induced debugging event.

Specific responses to significant events are listed in Table 6.

Assessing Your Location After an Event

If you are uncertain about your current program location after an event occurs, you can list the current frame by using the `frame` command, or the `select-frame` command as follows:

```
(eInspect 6,679):frame
#0  test_complexTypes() () at \SYS03.$D007.SYMBAT1.SCXXTST:424
424       printf( "%s test_complexTypes\n", getStepPrefix( 1 ) );
```

Table 6 Event Handling by Native Inspect

Events	Response by Native Inspect

Breakpoint events
`Instruction breakpoint`
`Instruction step`
`TNS instruction breakpoint`
`MAB`
`Process entered event`
`Process entered debug event`
	Displays current code location (PC) and prompts user for input. (For TNS code breakpoint, displays the current native code location.)

Entering debug events
`Process created in debug`
`Process puts itself in debug`
`Process forced into debug`
`Process executes embedded`
`Breakpoint (TNS process)`
	Creates a debug session with the process (attaches to the process); makes the process the current process; displays the current code location and prompts the user for input. (For the embedded breakpoint in a TNS process, displays the native code location.)

Signal event `Process signalled`	Responds according to the preferences set with the `mh` command. Actions include handing control to the user, printing a message, and forwarding the signal to the program. (Native Inspect gains control of the current process at the point of signal generation, before signal handler setup code is invoked.)
Process death event `Process death`	Detaches from the current process and notifies you that the session was terminated because the current process died. If Native Inspect is debugging one or more additional processes, no current process is designated as the current process. If there are no other processes and Native Inspect was started automatically, Native Inspect stops. If you started the debugger, Native Inspect continues to prompt.
OSS exec event `OSS exec`	Receives `process death` event and terminates its session with the current process (the one that called `exec`). If the newly created process is running in the same CPU as Native Inspect and `tdm_execve` was used with the `debug` option, then Native Inspect receives the `process entered debug` event and attaches to the new process.

Switching Between Debuggers (Inspect and Visual Inspect)

If you need to use features that are unique to another debugger, you can switch to a different debugger as shown in Table 7.

Table 7 Commands for Switching Debuggers

Switch From	Switch To	Command
Native Inspect	Inspect or Visual Inspect (according to Debugger Selection Criteria (page 20))	`switch` command.
Visual Inspect	Native Inspect	**Switch to System Debugger** command in Visual Inspect (switches to Native Inspect).
	Inspect	`ADD PROGRAM` command in Inspect (receives process from Visual Inspect).
Inspect	Native Inspect	`SELECT DEBUGGER DEBUG` command in Inspect (transfers process to Native Inspect) or you can use the attach command in Native Inspect.
	Visual Inspect	**Open Program** command in Visual Inspect (receives the process from Inspect).

Breakpoint attributes are not passed between debuggers. When you switch debuggers, Native Inspect preserves the breakpoints that you have set in the current process, along with any conditions associated with the breakpoints, including ignore counts or disabling of a breakpoint. When you switch back to Native Inspect, the breakpoint conditions, ignore-counts, and disable attribute are reinstated just as they were when you switched control of the process.

When you switch to another debugger and return to Native Inspect the breakpoint status is as follows

- Breakpoint attributes are preserved for existing breakpoints.
- Breakpoints that you deleted using the other debugger are deleted from Native Inspect's breakpoint list.
- Breakpoints that you added using the other debugger are added to Native Inspect's breakpoint list, with default attribute values.

Stopping Native Inspect

If you started Native Inspect automatically to debug a process, (that is by means of the `RUND` command), Native Inspect runs as a separate process from the current process being debugged. It stops under the following conditions:

- When the current process stops, if Native Inspect was started automatically to debug a process, that is by means of the `RUND` command.
- When you enter the `exit` command or the `quit` command to explicitly stop. Native Inspect detaches from the current process and stops, leaving breakpoints in place. Note, however, that if any of these breakpoints is subsequently hit, another debugger instance is automatically started.

Differences Between Native Inspect and WDB and GDB

- WDB and GDB support a `run` command that is used to start a program from within the debugger. Native Inspect, however, does not allow you to start a process from within the debugger. You must start the process from a TACL or OSS prompt.

  You can, however, use the `attach` command to attach an instance of Native Inspect to a TNS/E native process. (Native Inspect also supports several commands, such as the `vector` command, that are not supported by WDB or GDB.)
- Native Inspect does not support deferred breakpoints. In WDB and GDB, deferred breakpoints can be set on functions before a program or library is loaded.
- You cannot call functions in the current process from Native Inspect.
- Native Inspect does not support threading (such as Standard POSIX Threads).
- Native Inspect supports debugging multiple processes, but WBD and GDB do not support multiprocess debugging in the way that Native Inspect does.

2 Using Native Inspect

Quick Start for Inspect Users

Table 8 lists the principal Native Inspect commands and their Inspect equivalents. This table is a useful cross-reference for users familiar with Inspect commands.

- For a more complete list of Inspect commands and equivalent Native Inspect commands, see Table 16 (page 136).
- For a list of Debug commands and their equivalent Native Inspect commands, see Table 15 (page 135).

Table 8 Principal Native Inspect Commands With Inspect Equivalents

Task	Native Inspect Command	Inspect Command
Program State Display Commands		
List program source	`list`	`source`
Trace stack frames	`bt`	`trace`
Select a stack frame	`frame [`*`frame-number`*`] *` [1]	`scope` *`number`*

Display a variable or evaluate an expression	`print` *`var`*
	or
	`print` *`expression`*
		`display` *`data-loc`*

Modify a variable	`set [`*`variable`*`]` *`expr value`*
	or
	`print` *`var=exp`*
		`m[odify]` *`data-loc`*

Execution Control Commands		

Step (over function calls) Step in (a function call) Step out (of the current function)	`next`
	
	`step`
	
	`finish`
		`step`
		
		`step in`
		
		`step out`

Set a code breakpoint	`b[reak]` *`locspec`*	`b[reak]` *`code-loc`*
Set a memory access breakpoint	`mab {`*`addr`*`|`*`var`*`}`	`b[reak]` *`data-loc`*
Delete a code breakpoint	`delete [`*`bkpt-num`*`]`	`clear` *`bkpt-num`*
Delete a memory breakpoint	`dmab`	`clear` *`number`*
Resume execution	`continue`	`r[esume]`
Machine State Commands		

Display registers	`info registers`
	or
	`info all-registers`
	or
	`info frame` *`number`*
		`d[isplay] registers all`

Display instructions	`disassemble`
	`[`*`addr`*`[`*`addr`*`]|`*`func`*`]`
		`icode` *`code-loc`*

[1] Native Inspect displays source and looks up variables relative to the selected frame.

Preparing to Debug Using Native Inspect

Before you can use Native Inspect for debugging, you need to do the following:

1. Compile your program files, and transfer them to the TNS/E host if necessary.
2. Gain debugging control of a process by using Native Inspect.
3. Load symbols information for the current process, if necessary.
4. Optionally, configure a search path for source files.

These steps are described in more detail in the following sections.

Compiling and Transferring Program Files

You have the following options for compilation:

- The TNS/E host by using a resident compiler.
- A PC by using the Windows cross-compiler for HP NonStop systems.

Compile your code with optimization level 0 or 1 (code compiled with optimization level 2 cannot readily be debugged). With optimization level 1, the following conditions apply:

- Statements might be deleted or merged.
- The debugger can display live data, but data no longer needed by the program might not be available.
- Values often reside in registers, with write-through to memory.
- The debugger displays only values that are known to be true.

If you are developing your program on a remote TNS/E system or on a Windows PC, transfer your files to the system where you will perform debugging. You will need the program files in addition to the DLLs you are using.

Gaining Control of a Process Using Native Inspect

To start a program under control of the debugger, use the TACL `RUND` command:

```
TACL 2> rund nitest
```

The debugger that is invoked by the `RUND` command is determined by a set of rules described in the section titled: Debugger Selection Criteria (page 20).

To debug a process that is running, use the TACL `DEBUG` command:

```
TACL 3> debug nitest , term $myterm
```

For complete information about other ways to gain control of a process using Native Inspect, and for additional examples, see Starting Native Inspect (page 19).

Optionally Loading Symbols Information

To debug a process using a symbolic debugger such as Native Inspect, symbols information must be loaded for the process you want to debug. When Native Inspect gains control of a process, it attempts to load symbols for the process.

You will need to explicitly load symbols if you want to debug the following:

- A loadfile whose symbols have been stripped (typically done on production systems to minimize file size). You must know the location of the identical version that contains symbols.
- Any DLLs that your program loads.

To load symbols, use the `symbol` command, `symbol-file` command, or `add-symbol-file` command:

```
symbol-file filename
```

For example, if you enter an unqualified file name, the file must exist in the current working directory:

```
(eInspect 0,380): symbol-file xvod02a
xvod02a: No such file or directory.
(eInspect 0,380): symbol-file $d0101.qagarth.xvod02a
Reading symbols from $d0101.qagarth.xvod02a...done.
```

You must enter separate symbol commands for all files of interest, using one `symbol` command to load the symbols for your program file, and separate symbol commands for each DLL.

Native Inspect automatically reads in symbol table information for DLL loadfiles that are loaded in response to the dlopen() system call. Similarly, the symbol table is automatically discarded if the DLL is unloaded using the dlclose() system call.

Understanding Global versus Per-Process Symbol Files

When Native Inspect loads a symbol file, by default the symbols are available only to the current process. Such a symbol file has per-process scope.

Specifying the `-g` (global) option when loading a symbol file gives the symbol file global scope. The symbols are then available to all processes being debugged by a single Native Inspect instance. Symbol files with global scope are useful for NonStop system symbols and for shared DLLs.

For example, if you want a symbol file to be associated with a shared DLL or library, the symbol file needs to have global scope. Include the `-g` (global) option in your `symbol-file` command or `add-symbol-file` command.

Any given symbol file can be loaded as both a global scope symbol file and a per-process scope symbol file.

Specifying a Load Address for Symbol Files

Symbols are read in and assigned addresses based on the actual load address of the corresponding loadfile, if it can be determined. Otherwise, the symbol addresses are based on the preferred load address of the loadfile argument.

To override the preferred load address, use the `add-symbol-file` command and specify the address at which the loadfile is loaded or where you expect it to be loaded. You should do this when a DLL is loaded at a different address than the preferred load address. The `info` command with the `dll` option displays the names of the loaded loadfiles along with related information, including the addresses at which they are loaded.

Considerations for Locating Symbols

- Native Inspect does consistency checking of symbol files. That is, if you load symbols from another file, Native Inspect checks versions or timestamps between the object file you specify and the object file being executed.
- When Native Inspect does not display source file and line number information for a stack frame, symbols are most likely unavailable for that frame.
- Native Inspect looks for symbols from symbol files associated with the current process. These symbol files can have global scope or per-process scope.

Understanding How Native Inspect Locates Files

When automatically started as the selected debugger, Native Inspect, like Inspect, uses your logon default subvolume (not your current working subvolume or directory) as its default subvolume. The current working directory contains the process being debugged.

You can reduce confusion by immediately entering a `cd` command and setting the default subvolume for Native Inspect to be your current working subvolume (that is, the subvolume where your source files are stored). In the following example, it is `$data1.mysubvol`):

```
(eInspect 0,380): cd $data1.mysubvol
```

Native Inspect also maintains a source search path for locating source files. To specify the location of your source, you can either use fully-qualified file names, or use the `dir` command to set a source search path to the correct directory. For more information, see Optionally Configuring a Search Path for Your Source Files (page 36).

Optionally Determining the Compilation-Time Source File Name

If you are debugging on a different system from the one used for compiling, Native Inspect cannot locate your source files at their originally compiled locations. For this reason, the `list` command will not be able to list your program source and will report an error:

```
(eInspect 0,384): list
Unable to open file \SIERRA.$YOSE.MHG2.NITEST
```

Then you should add the current subvolume to the search path for source files, as described in Optionally Configuring a Search Path for Your Source Files (page 36). For example:

```
(eInspect 0,384): dir $d0117.mysvol
Source directories searched: $d0117.mysvol:$cdir:$cwd
(eInspect 0,384): list
   35       char *new_ptr = "In print_and_break\n";
   36       int z = 7;
   37       printf ("About to call DEBUG'\n");
   38       DEBUG();
   39     }
   40     void  main (void) {
   41       char *local_ptr = "From main";
   42       int local_q = 0;
   43       call1(local_ptr,local_q);
   44     }
(eInspect 0,384):
```

Optionally Configuring a Search Path for Your Source Files

Loadfiles contain the original compiled locations of their source files. If you have moved your files between compilation and debugging, you must set a search path so that Native Inspect can locate your source files.

Changing Paths

If the unqualified file names of your source files have not changed between compilation and debugging, use the `dir` command to add entries to the directory search list that Native Inspect uses to locate source files. For example:

```
(eInspect 0,330): dir $myvol.mysubvol
```

Changing File Names

If the unqualified file names of your source files change between compilation and debugging (for example, when transferring source files from a PC to the system), use the `map-source-name` command to define mapping rules. For example:

`map-source-name` *`fully-qualified-pathname=file`* `|` *`new-path`*

To display the current search path for source files, use the `show` command with the `directories` option.

The search path you specify is global to all processes that you debug in a single session of Native Inspect.

NOTE: If you specify an unqualified source file name with the `map-source-name` command, Native Inspect automatically uses the directory search list to locate the file.

Advancing Execution to main() in C/C++ Programs

When you start a C/C++ program with a RUND (Guardian) or `run -debug` (OSS) command, execution, by default, automatically advances to `main()`, and you can then examine the state of your program at the beginning of the program. It is not necessary to set a breakpoint at `main()` to get control at that point.

You can change this default behavior by entering a `set` command (environment) with the `continue-to-main` option set to `off`. You should include this command in the `EINSCSTM` file (located in your default logon directory) so that it is executed during Native Inspect initialization.

If `set continue-to-main off` is specified, you must set a breakpoint at `main()` to stop execution and examine the state of your program at that point, as in the following example:

```
(eInspect 0,330): b main
C:\mywin\home\myfiles\test\nitest.c, line 40.
(eInspect 0,330): c
Continuing.
Breakpoint 1 main () at C:\mywin\home\myfiles\test\nitest.c:40
40       void main (void) {
(eInspect 0,330): bt
#0 main () at C:\mywin\home\myfiles\test\nitest.c:40
#1 0x70000e60:0 in_MAIN 90 at \SPEEDY.$DATA06.T8432H01.CPLMAINC:68000
(eInspect 0,330):
```

Specifying `set continue-to-main off` is useful for debugging global constructors in a C++ program.

Sample Native Inspect Session (C++ Program)

The sample session described in this section demonstrates the tasks listed in Table 9.

Table 9 Sample Native Inspect C++ Program Session – User Tasks

Task	Native Inspect Command
Set search path for source files	`dir`
List source	`list`
Set a code breakpoint	`break`
	`tbreak`
Trace stack frames	`bt`
Advance execution	`next`
	`nexti`
	`step`
	`stepi`
Display a variable	`print`
Modify a variable	`set`
End program and session	`kill`
Display memory in internal representation	`x`

Launching a C++ Program Under Native Inspect Control

The following example shows how to launch Native Inspect using the `RUND` command:

```
$DATA3 NITEST 30>rund nitest
TNS/E eInspect gdb Debugger [T1237 - 20-Dec-2011 16:43]
Copyright 2008 Free Software Foundation, Inc.
Copyright 2003-2012 Hewlett-Packard Development Company, L.P.

Native Inspect (based on GDB) is covered by the GNU General Public License.
Type "show copying" for conditions for changing and/or distributing copies.
Type "show warranty" for warranty/support information.

Working directory \PELICAN.$SYSTEM.STARTUP.
(Symbols read in for program loadfile \PELICAN.$DATA3.NITEST.NITEST.
Added process (3,591).
Switching process (3,591) to eInspect from DMON
Process (3,591) created using DEBUG option.

(eInspect 3,591):
```

Listing the Source

The following example shows an attempt to display the source code using the `list` command:

```
(eInspect 3,591):list
\PIPPIN.$D0117.NITEST.GARTCC: No such file or directory.
(eInspect 3,591):dir $data3.nitest
Source directories searched: $data3.nitest:$cdir:$cwd
(eInspect 3,591):list
69          char *new_ptr = "In print_and_break\n";
70          int z = 7;
71          printf ("About to execute 'break 0x247'\n");
72          DEBUG();
73        }
74        void  main(void)
75        {
75.1        char local_buf[80];
76          char *local_ptr1, *local_ptr2;
77          int local_q;
```

The output from the preceding example demonstrates the following:

- The source file could not be found at the location where it was compiled because the object file was moved to another system.
- Because the source file name did not change, the `dir` command is used to instruct Native Inspect to search the subvolume that contains a copy of the file.

Tracing the Stack

The following example shows a stack trace using the `bt` command:

```
(eInspect 3,591):bt
#0  main () at \PIPPIN.$D0117.NITEST.GARTCC:78
#1  0x700034d0:0 in _MAIN () at \SPEEDY.$RLSE.T8432H01.CPLMAINC:68
```

The output from the preceding example shows the entry for frame #0, identifying the current location where the program is suspended.

Controlling Execution

The following example shows execution control, using the `next`, `b`, and `c` commands:

```
(eInspect 3,591):next
   78.1                 strcpy( local_buf, "Hello world" );
(eInspect 3,591):next
   78.2                 local_ptr2 = local_buf;
(eInspect 3,591):b 79
Breakpoint 2 at 0x700012d0:1: file \PIPPIN.$D0117.NITEST.GARTCC, line 79.
(eInspect 3,591):c
Continuing.

Breakpoint 2, main () at \PIPPIN.$D0117.NITEST.GARTCC:79
   79                   local_q = 0;
```

The output from the preceding example shows the following:

- The `next` command steps execution over function, procedure, and program unit calls.
- You can set a breakpoint on any source line number.

Printing Variables and Memory

The following example shows how to print variables and display memory contents by using the `print` and `x` commands:

```
(eInspect 3,591):print local_ptr1
$1 = 0x80001f0 "From main"
(eInspect 3,591):print /x &local_ptr1[0]
$2 = 0x80001f0
(eInspect 7,911):print local_ptr2
$10 = 0x6ffffedc "Hello world"
(eInspect 7,911):x /4 local_ptr2
0x6ffffedc:     72 'H'       101 'e'       108 'l'       108 'l'
```

The output from the preceding example shows the following:

- Use of the `print` command to print the values of variables.

  NOTE: C/C++ character pointers are automatically dereferenced.

- Use of the `x` command to display memory in its internal representation.

Stepping Execution Into a Function

The following example shows how to step execution into a function by using the `next`, `step`, and `bt` commands:

```
(eInspect 3,591):next
   80                   call1(local_ptr1,local_q);
(eInspect 3,591):step
call1(char *, int) (string=0x0, q=0) at \PIPPIN.$D0117.NITEST.GARTCC:84
   84                 {
(eInspect 3,591):next
   85                   A1 a1;
(eInspect 3,591):bt
#0  call1(char *, int) (string=0x80001f0 "From main", q=0)
    at \PIPPIN.$D0117.NITEST.GARTCC:85
#1  0x70001320:0 in main () at \PIPPIN.$D0117.NITEST.GARTCC:80
#2  0x700034d0:0 in _MAIN () at \SPEEDY.$RLSE.T8432H01.CPLMAINC:68
```

The output from the preceding example shows the following:

- Use the `step` command to step execution into a function.
- Parameter values are not available until execution is stepped through function prolog code.

Setting a Memory Access Breakpoint (MAB)

The following example shows how to set a memory access breakpoint (MAB) by using the `mab`, `c`, and `print` commands:

```
(eInspect 3,591):mab structure.a
No symbol "structure" in current context.
(eInspect 3,591):next
   86      printf("%s q = %d\n",string,q);
(eInspect 3,591):print a1
$3 = {
  a = 0x0,
  b = 0
}
(eInspect 3,591):mab a1.b

(eInspect 3,591):c
Continuing.
About to execute 'break 0x247'
Process (3,591) called DEBUG.
0x70001190:0 in print_and_break() () at \PIPPIN.$D0117.NITEST.GARTCC:72
   72      DEBUG();
(eInspect 3,591):c
Continuing.
```

```
Process (3,591) received DS_EVENT_MAB (seg:65535, addr:0x6FFFFE44, pc:0x70002862
, len:4 type:1)
A1::func(char *, int) (this=0x6ffffe40, string=0x8000220 "From call1", c=1)
    at \PIPPIN.$D0117.NITEST.GARTCC:29
   29              }
(eInspect 3,591):print *this
$5 = {
  a = 0x8000220 "From call1",
  b = 1
}
```

The output from the preceding example shows the following:

- Use of the `mab` command to set a memory access breakpoint on a scalar variable.
- Before the memory access breakpoint is hit, the program calls `DEBUG`, after which execution is resumed.

Stopping Mid-Statement

There are situations, for example stopping at a MAB, when the debugger is not stopped at the starting instruction of a source statement. Native Inspect shows this "mid-statement" position by prefixing an instruction address:

```
0x70001190:0 in print_and_break() () at \PIPPIN.$D0117.NITEST.GARTCC:72
```

Ending the Program and Debugging Session

The following example shows how to end the session by using the `kill` command:

```
(eInspect 3,591):kill
Kill the current process? (y or n) y
Process (3,591) exited with code 06.
Removed process (3,591).
eInspect is exiting...
Killed
TACL>
```

Sample Native Inspect Session (COBOL Program)

The sample session in this section illustrates the tasks shown in Table 10.

Table 10 Sample Native Inspect COBOL Program Session – User Tasks

Tasks	Native Inspect Command
Advance execution	`next`
	`nexti`
	`step`
	`stepi`
Set a code breakpoint	`break`
	`tbreak`
List source	`list`
Display a variable	`print`
Modify a variable	`set`
Resume execution	`continue`
End program and session	`kill`

Starting a Program Under Native Inspect Control

The following example shows how to start the session by using the `RUND` and `next` commands:

```
DATA3 COBBAT 53> rund xcs000ds0
TNS/E eInspect gdb Debugger [T1237 - 20-Dec-2011 16:43]
Copyright 2008 Free Software Foundation, Inc.
Copyright 2003-2012 Hewlett-Packard Development Company, L.P.

Native Inspect (based on GDB) is covered by the GNU General Public License.
Type "show copying" for conditions for changing and/or distributing copies.
Type "show warranty" for warranty/support information.

TWorking directory \PELICAN.$SYSTEM.SYSTEM.
Symbols read in for program loadfile \PELICAN.$DATA3.COBBAT.XCS000D0.
Added process (3,1012).
Switching process (3,1012) to eInspect from DMON
Breakpoint 1 at 0x70001600:0: file \PELICAN.$DATA3.COBBAT.SCS000D, line 5.
(eInspect 3,1012):next
318           perform initialization thru initialization-exit.
Current language:  auto; currently COBOL
```

The `next` command advances execution from the beginning of the program unit to the first executable statement.

Listing Source and Setting a Breakpoint at a Line Number

The following example shows how to list source and set a breakpoint at a line number by using the `list` and `break` commands:

```
eInspect 3,1012):list
  313
  314  PROCEDURE DIVISION.
  315
  316  MAIN SECTION.
  317  CALE-1-MAIN-PARA-1.
  318    PERFORM INITIALIZATION THRU INITIALIZATION-EXIT.
  319    PERFORM READ-SEQ1-TABLE THRU SEQ1-TABLE-EXIT 4 TIMES.
  320    IF ERROR-DETECTED
  321       PERFORM STOP-RUN.
  322    PERFORM READ-SEQ2-TABLE UNTIL EOF OR ERROR-DETECTED.
(eInspect 3,1012):break 319
Breakpoint 2 at 0x70001bd0:0: file \PELICAN.$DATA3.COBBAT.SCS000D, line 319.
```

The `break` (or abbreviated `b`) command sets a breakpoint at line 319 to suspend execution upon return from the first `PERFORM` invocation.

Stepping Execution

The following example shows how to step through execution by using the `step` and `next` commands:

```
eInspect 3,1012):step
  351   PERFORM OPEN-SEQ1-FILE.
(eInspect 3,1012):next
  352   IF SEQ1-FILE-ER THEN
```

The preceding example demonstrates the following:

- The `step` command steps execution into `PERFORM` invocations and calls to program units.
- The `next` command steps execution over `PERFORM` invocations and calls to program units.
- `PERFORM` invocations are not listed on the call stack.

Displaying a Level 88 Condition Name

The following example shows how to display a level 88 condition name by using the `print` and `continue` commands:

```
77 SEQ1-FLAG PIC 9.
   88 SEQ1-FILE-ER     VALUE 0.
   88 SEQ1-FILE-OPENED VALUE 1.

(eInspect 3,1012):print seq1-file-er
$1 = F
(eInspect 3,1012):print seq1-flag
$2 = 1
(eInspect 3,1012):print seq1-file-opened
$3 = T
(eInspect 3,1012):continue
```

Examining a Record

The following example shows how to examine a record by using the `next`, `print`, and `p` commands:

```
Breakpoint 2, CALE-1 () at \PELICAN.$DATA3.COBBAT.SCS000D:319
  319    PERFORM READ-SEQ1-TABLE THRU SEQ1-TABLE-EXIT 4 TIMES.

(eInspect 3,1012): next
  320       IF ERROR-DETECTED

(eInspect 3,1012): next
  322          PERFORM READ-SEQ2-TABLE UNTIL EOF OR ERROR-DETECTED.

(eInspect 3,1012): print seq2-rec

$4 =
  NAME = "\000\000\000\000\000\000\000"
  FILLER = ""
  WORKS = "\000\000\000\000\000\000"
  FILLER = ""
  REST-OF-LINE = '\000' <repeats 33 times>

(eInspect 3,1012): next
  323        IF ERROR-DETECTED

(eInspect 3,1012): print name
Reference to NAME is not unique.
Add more qualification to disambiguate the reference.

(eInspect 3,1012): print name of seq2-rec
$5 = "JStrauss"

(eInspect 3,1012): print rest-of-line
$6 = "Gypsy Baron  1881Fledermouse  1879"

(eInspect 3,1012): print seq2-rec.works
$5 = "opera04"

(eInspect 3,1012): print seq2-rec
$7 =
  NAME = "JStrauss"
  FILLER = " "
  WORKS = "opera04"
  FILLER = " "
  REST-OF-LINE = "Gypsy Baron  1881Fledermouse  1879"

(eInspect 3,1012): p rtq1 in reclast
$11 = "RECL.R231"

(eInspect 7,320):p rtq4 in rtq3 in rtq2 in rtq1(4,2,3,5)
$51=
RTQ5="\100"
RTQ6="\100"
RTQ6="\000)"
```

```
(eInspect 3,1012): p rtq6 of rtq3 in rtq1(4,2,3,4,2)
$67="GH"
```

The preceding example demonstrates the following:

- Record field names that are unique do not require qualification. Native Inspect reports an error if the name is not unique.
- You can qualify field names using COBOL OF or IN syntax, or the Native Inspect period (.) syntax.

Modifying a Record Field

The following example shows how to modify a record field by using the `set` command:

```
(eInspect 3,1012):set name of seq2-rec  = "Bach"
(eInspect 3,1012):print seq2-rec
$4 =
  NAME = "Bach    "
  FILLER = " "
  WORKS = "opera04"
  FILLER = " "
  REST-OF-LINE = "Gypsy Baron  1881Fledermouse  1879"
```

The preceding example demonstrates the following:

- Values are padded or truncated according to COBOL rules.
- Native Inspect ignores any JUSTIFIED clause on a data item.

Examining Tables

The following example shows how to examine tables by using various commands:

```
(eInspect 3,1012): print recording-data
$3 =
  WHO = (
      NAME = "Puccini "
      WORKS = (
          TITLE = "Sigfried     "
          LISTING-INFO =
            CONDUCTOR = "Solti      "
            PRICE = "$34,567.00"
          TITLE = "Boheme       "
          LISTING-INFO =
            CONDUCTOR = "Walter     "
            PRICE = "$34,572.00"
      NAME = "Massenet"
      WORKS = (
          TITLE = "Otello       "
          LISTING-INFO =
            CONDUCTOR = "Davis      "
           PRICE = "$34,577.00"
          TITLE = "Manon        "
          LISTING-INFO =
            CONDUCTOR = "Rudel      "
            PRICE = "$34,582.00"
      NAME = "Handel  "
      WORKS = (
          TITLE = "Arabella     "
          LISTING-INFO =
            CONDUCTOR = "Solti      "
            PRICE = "$34,587.00"
          TITLE = "Cemele       "
          LISTING-INFO =
            CONDUCTOR = "Walter     "
            PRICE = "$34,592.00"
            NAME = "Bellini "
      WORKS = (
          TITLE = "Semiramide   "
          LISTING-INFO =
            CONDUCTOR = "Davis      "
            PRICE = "$34,597.00"
          TITLE = "Norma        "
          LISTING-INFO =
            CONDUCTOR = "Rudel      "
            PRICE = "$34,602.00"
(eInspect 3,1012): print title(1,2)
Reference to TITLE is not unique.
Add more qualification to disambiguate the reference.
(eInspect 3,1012): print title of works of who of recording-data (1,2)
$5 = "Boheme       "
(eInspect 3,1012): print recording-data.who.works.title (1,2)
$7 = "Boheme       "
eInspect 3,1012): print works of who of recording-data (1)
$6 = (
    TITLE = "Sigfried     "
    LISTING-INFO =
      CONDUCTOR = "Solti      "
    PRICE = "$34,567.00"
    DEALERS-CODE = 34572,
    TITLE = "Boheme       "
    LISTING-INFO =
      CONDUCTOR = "Walter     "
    PRICE = "$34,572.00"
    DEALERS-CODE = 34577)
(eInspect 1,301): ptype rec5
type = RECORD
R51 RECORD OCCURS 2 TIMES
R511 PIC X(16)
R512 PIC X(16)
R513 RECORD OCCURS 3 TIMES
R5131 PIC XXX
R5132 PIC XXX
(eInspect 1,301): p rec5
$40 =
R51 = (
R511 = "REC5.R51[1].R511"
R512 = "REC5.R51[1].R512"
R513 = (
R5131 = "111"
R5132 = "112",
R5131 = "111"
R5132 = "112",
R5131 = "111"
R5132 = "112"),
R511 = "REC5.R51[1].R511"
R512 = "REC5.R51[1].R512"
R513 = (
R5131 = "211"
R5132 = "212",
R5131 = "211"
R5132 = "212",
R5131 = "211"
R5132 = "212"))
(eInspect 1,301): p r513(2)
$43 = (
R5131 = "211"
R5132 = "212",
R5131 = "211"
R5132 = "212",
R5131 = "211"
R5132 = "212")
(eInspect 1,301): p r5132(1,2)
$45 = "112"
(eInspect 2,924): p r5132(2,3)
$8 = "212"
(eInspect 1,305): ptype rec-a
type = RECORD
ITEM-A PIC 99
DATA-A PIC X OCCURS ITEM-A (MAX:10) TIMES
(eInspect 1,305): #
(eInspect 1,305): # Let's set DEPENDING ON parameter, i.e. "ITEM-A", to 5
(eInspect 1,305): #
(eInspect 1,305): set rec-a.item-a = 5
(eInspect 1,305): p rec-a
$4 =
ITEM-A = 05
DATA-A = ("A", "A", "A", "A", "A")
(eInspect 1,305): set rec-a.data-a(4) = "I"
(eInspect 1,305): p rec-a
$4 =
ITEM-A = 05
DATA-A = ("A", "A", "A", "I", "A")
```

The preceding example demonstrates the following:

- You can apply subscripts to an unqualified record element name as long as the name is unique.
- Subscripts are separated by commas and must always occur in the last item of the reference.

Setting a Breakpoint on a Nested Program Unit

The following example shows how to set a breakpoint on a nested program unit by using the `b`, `c`, `bt`, and `next` commands:

```
(eInspect 3,1012):b cale-1.cale-1-1
Breakpoint 3 at 0x7000c200:2: file \PELICAN.$DATA3.COBBAT.SCS000D, line 588.
(eInspect 3,1012):c
Continuing.

Breakpoint 3, CALE-1.CALE-1-1 (ID-1=1, ID-2=10, ID-3=100,
    ID-4=1000, ID-5=10000, ID-6=1000000)
    at \PELICAN.$DATA3.COBBAT.SCS000D:588
   588  PERFORM CALE-1-1-TEST-1
(eInspect 3,1012):bt
#0  CALE-1.CALE-1-1 (ID-1=1, ID-2=10, ID-3=100,
    ID-4=1000, ID-5=100000, ID-6=1000000)
    at \PELICAN.$DATA3.COBBAT.SCS000D:588
#1  0x70008d00:0 in CALE-1 () at \PELICAN.$DATA3.COBBAT.SCS000D:437
(eInspect 3,1012):next
  589  PERFORM CALE-1-1-TEST-2
```

The preceding example demonstrates that you must qualify the nested program unit name with the name of the enclosing program unit or units.

Debugging Copy Libraries

The following example shows how to debug copy libraries by using the `list`, `b`, `c`, and `next` commands:

```
(eInspect 3,1012):list
  310     Test-SingleInclusion.
  311       DISPLAY " ".
  312       DISPLAY "Test-SingleInclusion: Begin".
  313     ?SOURCE CLIBCLB1( RANGE-110-112 )
  314       DISPLAY " ".
  315       DISPLAY "Test-SingleInclusion: End".
(eInspect 3,1012):list CLIBCLB1:110
  105     02 ALPHA-A26     PIC A(26) VALUE "abcdef".
  106
  107
  108
  109     ?SECTION RANGE-110-112
  110        DISPLAY "  CLIBCLB1: RANGE-110-112: line 110".
  111        DISPLAY "  CLIBCLB1: RANGE-110-112: line 111".
  112        DISPLAY "  CLIBCLB1: RANGE-110-112: line 112".
  113
  114
(eInspect 3,1012):b CLIBCLB1:110
Breakpoint 2 at 0x700063b0:0: file \YOSQA1.$DATA1.SHCOBAT.CLIBCLB1, line 110.
Breakpoint 3 at 0x70006f30:0: file \YOSQA1.$DATA1.SHCOBAT.CLIBCLB1, line 110.
Breakpoint 4 at 0x70007770:0: file \YOSQA1.$DATA1.SHCOBAT.CLIBCLB1, line 110.
(eInspect 3,1012):c
Continuing.

Breakpoint 2, COPYLIB-STATEMENTS () at \YOSQA1.$DATA1.SHCOBAT.CLIBCLB1:110
  110     DISPLAY "  CLIBCLB1: RANGE-110-112: line 110".
(eInspect 3,1012):list
  105              02 ALPHA-A26     PIC A(26) VALUE "abcdef".
  106
  107
  108
  109     ?SECTION RANGE-110-112
  110        DISPLAY "  CLIBCLB1: RANGE-110-112: line 110".
  111        DISPLAY "  CLIBCLB1: RANGE-110-112: line 111".
  112        DISPLAY "  CLIBCLB1: RANGE-110-112: line 112".
  113
  114
(eInspect 3,1012):next
  CLIBCLB1: RANGE-110-112: line 110
  111        DISPLAY "  CLIBCLB1: RANGE-110-112: line 111".
(eInspect 3,1012):next
  CLIBCLB1: RANGE-110-112: line 111
  112        DISPLAY "  CLIBCLB1: RANGE-110-112: line 112".
(eInspect 3,1012):next
  CLIBCLB1: RANGE-110-112: line 112
  314        DISPLAY " ".
```

The preceding example demonstrates the following:

- The `list` command, by default, lists source from the file that contains the statement at which execution is suspended. It lists one file at a time and does not show the contents of included files.
- You can view the contents of an included file by specifying the base file name and the line number relative to the file you want to view.
- Specify the base file name and line number to set a breakpoint on a line in a copy library. Native Inspect sets breakpoints at each program location where that copy library line is included. Therefore, multiple breakpoints might be reported for each copy library line. Clear any breakpoints that you do not need.

Terminating the Debugging Session

Use the `kill` command to terminate the process and debugging session:

```
(eInspect 3,1012):kill
Kill the current process? (y or n) y
Process (3,1012) exited with code 06.
Removed process (3,1012).
eInspect is exiting...
Killed
TACL>
```

If you want the process to continue execution, use the `continue` command followed by the `exit` command to terminate the debugging session.

3 Using Native Inspect With COBOL Programs

This section describes concepts and additional details for using Native Inspect with COBOL programs.

Understanding how Native Inspect finds Data Items

Native Inspect follows COBOL scoping rules for finding data items specified in Native Inspect commands. That is, Native Inspect attempts to find the item in the current program unit (the program unit where execution is currently suspended). If Native Inspect cannot find the item in the current program unit, it looks in containing program units if the data item is declared GLOBAL.

In nested program units, a data item declared in an inner program unit can “hide” a global data item with the same name that is also declared in an outer program unit. When execution is suspended in the inner program unit, the only way to access the item in the outer program unit is to change the currently selected stack frame to the stack frame containing the outer item.

Handling of SOURCE and COPY Directives

Displaying Lines Included by SOURCE and COPY Directives

Native Inspect does not merge the source lines from SOURCE and COPY directives into the source listed for a program unit. Therefore, when you enter a `list` *`filename`*`:`*`line-number`* command, only the lines in the specified file are shown, and not the lines included by any `SOURCE` or `COPY` directives in the file. To list any source lines that were copied by a nested `SOURCE` or `COPY` directive, you must enter a separate list command for the file named in the directive.

Setting Breakpoints at Lines Included by SOURCE and COPY Directives

In a COBOL program that contains SOURCE and COPY directives, identical line numbers can occur within a program unit. If, in a break command, you specify a line number with no qualification, Native Inspect sets a breakpoint at each instance of that line number. To set a breakpoint at a particular instance of a line number, you must qualify the line number with the name of the containing file, using the `filename:`*`line-number`* notation.

Displaying Source Lines

In a program consisting of multiple files, Native Inspect handles the `list` command as follows:

- If the `list` command specifies a file name, Native Inspect lists the contents of that file.
- If the `list` command does not specify a file name, Native Inspect lists the contents of the last source file listed since execution was last suspended.
- If no source file was listed since execution was last suspended, Native Inspect displays the contents of the file where execution is currently suspended.

NOTE: Note the following conditions:

- Native Inspect does not expand `SOURCE` and `COPY` directives to list their source inline. To display source lines included by `SOURCE` and `COPY` directives, use a separate list command, as described under Displaying Lines Included by SOURCE and COPY Directives (page 47).
- The `list` command does not display the results of string substitution done by a `REPLACE` statement or a `REPLACING` clause.

Specifying Variables and Tables

Specifying Variables

To reference a variable that is a unique member of only one record in a COBOL program, you simply specify the variable name. For example, consider the following declaration in a COBOL program:

```
01 REC.
   02 VAR NATIVE-2.
   02 STR PIC X(9).
```

As long as the variables are unique within the program unit, you can specify the variable VAR and STR in any Native Inspect command that accepts a variable name. However, if a variable is not unique within the program unit (that is, if the same variable name is declared in more than one record), you must qualify the variable name. Qualify the record name by using the COBOL reserved word as follows:

- `IN`.
- `OF`.
- Period (`.`) syntax.

For example, consider the following declaration:

```
01 REC.
   02 INT PIC X(10).
   02 VAR PIC X(10).
   02 REC2.
      03 INT PIC X(13).
      03 VAR2 PIC X(14).
```

The variable INT is defined in both REC and REC2. You would specify the variable INT that is contained in REC2 in any of the following ways:

```
INT OF REC2
INT IN REC2
REC2.INT
```

When you use the period (`.`) syntax, the record name comes first, followed by the variable name. (This use of the period is Native Inspect syntax, not COBOL syntax.)

When qualifying a non-unique variable name, you must specify only enough qualifiers to make the variable reference unique. Thus, in the preceding declaration, you need not specify any of the following:

- `INT OF REC2 OF REC`.
- `INT IN REC2 IN REC`.
- `REC.REC2.INT`.

If you specify a non-unique variable name without qualification or with insufficient qualification, Native Inspect reports an error.

Specifying Tables

You reference individual table elements by specifying the table name followed by subscripts in parentheses. For multidimensional tables, the subscripts are separated by commas. For example, consider the following declaration:

```
02 TOTAL OCCURS 20 TIMES.
   03 TOTAL-A OCCURS 3 TIMES.
```

The following example refers to a specific element of the two-dimensional table `TOTAL-A`:

```
TOTAL-A (4, 2)
```

In Native Inspect, unlike in a COBOL program, you cannot use spaces as subscript separators.

When referencing a multilevel table in Native Inspect, you can apply subscripts only to the last item in the reference. For example, consider the following declaration:

```
01 VEHICLE.
   03 MODEL OCCURS 9 TIMES.
      05 STYLE OCCURS 12 TIMES.
          07 COLOR OCCURS 15 TIMES PICTURE 9(10).
```

Examples of valid table references are as follows:

```
MODEL (3)
STYLE OF MODEL (3,11)
COLOR OF STYLE OF MODEL (3,11,14)
MODEL.STYLE.COLOR (3,11,14)
```

You reference an entire table by specifying the table name without subscripts. For example, the following command displays all elements of table `TOTAL-A`:

```
(eInspect 3,1012): print TOTAL-A
```

Subscripts are used to refer to specific instances of scalar variables that are part of a table of records. For example, consider the following declaration:

```
01 MASTER.
   02 TABLE-1 OCCURS 5 TIMES.
      03 TABLE-2 OCCURS 5 TIMES.
         04 ELEMENT PIC X.
```

The following is an example of a reference to an instance of the scalar variable `ELEMENT`:

```
ELEMENT (3, 2)
```

This use of the comma is Native Inspect syntax, not COBOL syntax.

Specifying Tables With Variable Upper Bounds

You can declare COBOL tables with a variable upper bound. For example:

```
01 MASTER.
   03 TABLE OCCURS 5 TO 10 TIMES DEPENDING ON ITEM.
```

If, in a Native Inspect print command, you specify the table name with no subscripts to print the entire table, Native Inspect evaluates the variable upper bound specified with the DEPENDING ON keywords to determine the number of elements to display. However, in rare situations where Native Inspect is unable to evaluate the variable upper bound, the table is displayed up to the maximum specified by the OCCURS keyword.

For the preceding declaration, if you specify the following, then Native Inspect attempts to evaluate the variable `ITEM` to determine the number of elements to display:

```
print TABLE
```

Specifying Level 88 Condition Names

You can attach a condition name to any data item. You specify a condition name the same way as you would specify a member of a record. For example, consider the declaration:

```
01 REC.
   05 ITEM PIC 99.
      88 GOOD-ITEM VALUE 10 THRU 20.
```

You could specify the following:

```
GOOD-ITEM OF ITEM OF REC
```

Alternatively, omit any qualifiers not required for uniqueness, just as you would for any other data item.

You can also attach a condition name to a table or to a member of a record that is a table. In that case, you must specify a subscript with the condition name. To evaluate the condition name, specify the same subscript on the condition name as you would for the condition variable name. For example, consider the following declaration:

```
01 REC.
   02 TABLE OCCURS 4 TIMES.
      03 ITEM PIC 99.
          88 ITEM-OK VALUE 12.
```

Any reference to `ITEM-OK` requires a subscript because any reference to `ITEM` requires a subscript. For example, the following command evaluates and displays the second instance of `ITEM-OK`:

```
(eInspect 3,1012):print ITEM-OK(2)
```

Displaying Variables

Native Inspect follows COBOL rules for displaying numeric, alphanumeric, and edited data items. Considerations are:

- Native Inspect does not allow the use of the `PICTURE` clause to format variables for display.
- You can display a variable in a different radix by using the `FORMAT` clause of the `print` and `x` commands, as described in the section:Performing Machine-Level Debugging (page 54).

Displaying Level 88 Condition Names

Native Inspect displays Level 88 condition names as one of the values ‘`T`’ or ‘`F`’. The value displayed depends on the value of the variable to which the condition belongs. For example, consider the following declaration:

```
77 VAR PIC S99 VALUE 1.
   88 COND-1 VALUE 1.
   88 COND-2 VALUE -1.
```

The following Native Inspect commands display the values indicated:

`print COND-1`	displays the value ‘`T`’ because the value of COND-1 matches the value of VAR.
`print COND-2`	displays the value ‘`F`’ because the value of COND-2 does not match the value of VAR.
`print VAR`	displays the actual value stored in VAR; in this case the value is `1`.

Displaying Argument Values

By default, Native Inspect displays only the addresses of program and function arguments when a breakpoint is encountered, a backtrace is done, or execution steps into a function. To display the actual argument values on the occurrence of any of these events, specify the following command:

```
(eInspect 3,1012): set print cobol-arg-values on
```

Displaying Unprintable Characters

COBOL support is improved to display unprintable characters and substring searches. In previous releases, Native Inspect would display unprintable characters using C programing format. For example, a character with a value of `0` was displayed as `\000`. In this release, unprintable characters are printed using COBOL Hexadecimal Nonnumeric Literals. For example, a character with a value of 0 now displays as follows: `X”00”`.

Displaying the Length of the COBOL Variables

You can display the length of the COBOL variables using the `print length` command:

print length *cobol-variable* (or) print length (*cobol-variable*)

To print the length of an individual table element, subscript the table element. For example, consider the following declaration:

```
01 MASTER.
     02 TABLE-1 OCCURS 5 TIMES.
```

```
          03 TABLE-2 OCCURS 5 TIMES.
                04 ELEMENT PIC X.

(eInspect 1,821):print length MASTER
$2 = 25
(eInspect 1,821):print length (MASTER.TABLE-1(1))
$3 = 5
(eInspect 1,821):print length TABLE-2(1,1)
$4 = 1
```

For tables with a variable upper bound, the maximum length of the table is displayed irrespective of the value of the data item specified in the DEPENDING ON clause. For example, consider the following declaration:

```
 01 MASTER.
      03  ITEM PIC 99 VALUE 7.
           03  TABLE4 OCCURS 5 TO 10 TIMES DEPENDING ON ITEM.
                 05 ITEM-5 PIC 99.

 (eInspect 1,807):print length MASTER
 $6 = 22
```

Handling of REDEFINES and RENAMES

Native Inspect treats variables named in a REDEFINES or RENAMES clause the same as other variables in the record. Thus, when Native Inspect displays a record, the REDEFINE and RENAME variables are also shown. For example, consider the following declaration:

```
01 REDEF.
   02 NAME PIC A(30).
   02 OTHER-NAME REDEFINES NAME.
      03 FIRST-NAME PIC A(20).
      03 LAST-NAME PIC A(10).
         66 FNAME RENAMES FIRST-NAME.
         66 LNAME RENAMES LAST-NAME.
```

The `print REDEF` command displays all items in the record, including `OTHER-NAME`, `FNAME`, and `LNAME`, as follows:

```
  NAME=...
   OTHER-NAME=
      FIRST-NAME=
      LAST-NAME=...
   FNAME=...
   LNAME=...
```

Assigning Values to Data Items

Assigning Values to Variables

Native Inspect provides two ways of assigning values to data items:

- Using the `set` command.
- Using the `print` *`data-item`* `=` *`value`* command.

Native Inspect follows COBOL rules for assigning values to numeric and alphanumeric variables.

Some considerations are:

- Native Inspect follows COBOL rules for truncation and padding, but ignores the JUSTIFIED clause.
- When assigning a numeric value to an alphanumeric item, the numeric item is treated as an alphanumeric item with digits moved from left to right.
- Native Inspect does not allow assignment to EDITED data items.

- If the name of a variable is the same as a Native Inspect option recognized by the `set` command, you must use either the `print` command with the assignment operator or the variable clause of the `set` command to assign a value to the variable.
- The maximum size of a numeric literal is 18 decimal digits, 16 hexadecimal digits, or 22 octal digits.

Changing the Radix of Numeric Literals

The default radix of a numeric literal is base 10. You can use the following notations to change the value of a numeric literal:

- `0xvalue` – Specifies a hexadecimal value.
- `0number` – Specifies an octal value.

Considerations when Changing the Radix

- Native Inspect does not support the COBOL syntax for hexadecimal literals (`H`"*`value`*").
- You cannot change the radix of an alphanumeric literal.

Assigning Values to Level 88 Condition Names

You cannot modify a level 88 condition name. For example, consider the following declaration:

```
01 REC.
   02 TABLE OCCURS 4 TIMES.
      03 ITEM PIC 99.
          88 ITEM-OK VALUE 12.
```

You cannot assign a new value to `ITEM-OK`. You can change only the value of the underlying data item, in this case, the level 03 item `ITEM`.

Assigning Values to Tables and Records

You can assign multiple values to items that compose a table or record by specifying a list of values separated by commas and enclosed in braces (`{}`). For example, consider the following declaration:

```
01 MASTER.
   02 TABLE-1 OCCURS 2 TIMES.
      03 TABLE-2 OCCURS 3 TIMES.
         04 ELEMENT PIC X.
```

The following command assigns values to all six occurrences of ELEMENT:

```
(eInspect 3,1012): set ELEMENT={2, 5, 1, 8, 0, 4}
```

The values of the elements of ELEMENT are as follows:

- `ELEMENT(1,1) = 2`
- `ELEMENT(1,2) = 5`
- `ELEMENT(1,3) = 1`
- `ELEMENT(2,1) = 8`
- `ELEMENT(2,2) = 0`
- `ELEMENT(2,3) = 4`

You can nest table and record assignment by using multiple pairs of braces, as shown in the next example. Assume the following declaration:

```
01 rec6.
   02 array Native-2 occurs 10 times.
   02 another2 Native-2.
```

The following command assigns and prints the two tables in record `rec6`:

```
(eInspect 1,1032): print rec6={{1,2,3,4,5,6,7,8,9,10}12}
$1 =

ARRAY = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
ANOTHER2 = 12
```

If, in an assignment, you specify more values than there are elements in the table, an error is generated. If you specify fewer values, the assignment terminates when all values in the list are used.

You cannot assign a value to an entire table or record by specifying a single value. For example, assuming the preceding declaration, the following command generates an error:

```
eInspect n,n:set ELEMENT=0
```

The following command sets only `ELEMENT (1,1)` to zero:

```
eInspect n,n:set ELEMENT={0}
```

Assigning Values to Character Strings

The following syntax is supported by Native Inspect:

```
01 REC.
   02 MYSTRING PIC X(10).
```

The following command assigns a value to the entire string `MYSTRING`:

```
(eInspect 3,1012): set MYSTRING="Hello!!!!!"
```

If, in this example, you specify the following command, the character string is padded with spaces on the right according to the same rules used by the COBOL `MOVE` statement:

```
(eInspect 3,1012): set MYSTRING="Hello"
```

The following command prints 2 characters starting at index 1:

```
(eInspect 3,1012): print MYSTRING(1:2)
```

The following command prints a substring of `MYSTRING` starting at index 2 until the end:

```
(eInspect 3,1012): print MYSTRING(2:)
```

The following command modifies two characters:

```
(eInspect 3,1012): print MYSTRING(1:2) = "XX"
```

Native Inspect supports the COBOL string concatenation operator (&). For example, assume the following declaration:

```
01 REC
   02 ASTRING PIC X(4).
```

The following commands assign a value to `ASTRING` using concatenation, then displays the string value:

```
(einspect 0,434):set ASTRING “AB” & “CD”
(einspect 0,434):print ASTRING
$6 = “ABCD”
```

Evaluating Expressions

Native Inspect supports COBOL arithmetic expressions and a subset of COBOL conditional expressions, and follows COBOL rules for evaluating these expressions as follows:

- Arithmetic operators are as follows:

  ```
  +
  -
  *
  /
  **
  ```

- Conditional expressions use the following operators:

```
GREATER
NOT GREATER
LESS THAN
NOT LESS THAN
EQUAL TO
NOT EQUAL TO
```

- Conditional expressions use the following logical operators:

  ```
  AND
  OR
  ```

- The following symbols are supported:

  ```
  >
  <
  <>
  <=
  >=
  ```

If a variable is specified as the result of an expression, Native Inspect stores as much of the result as possible as allowed by the variable's type.

Native Inspect does not allow the use of the following in expressions:

- Intrinsic functions.
- Class conditions.
- Sign conditions.
- Switch-status conditions.
- Abbreviated conditions.

Displaying Data Item Types

You display the type of a COBOL data item in Native Inspect by using the `ptype` and `whatis` commands. Wherever possible, types are displayed as declared in the COBOL program, with these exceptions:

- Level 88 condition names are shown as type `bool`, since there is no corresponding COBOL type.
- The PICTURE string shown for edited items is `PIC X(`*`length`*`)`.

Performing Machine-Level Debugging

Native Inspect provides a full set of commands for machine-level debugging. These commands enable you to perform such low-level tasks as examining memory, examining registers, and listing machine-level instructions. One such command, the x command, provides a special form for use with COBOL programs. The x command, when used with the `ADDRESS OF` option, is useful for generating memory dumps. This command has the following form:

```
x /format ADDRESS OF variable
```

where *`format`* specifies a repeat count, the format to use, and the size of the variable to be examined; and *`variable`* specifies the starting address of the memory dump.

For more information about the `x` command, see x Command (page 125)

Controlling Execution

You control execution of a program by setting breakpoints at locations in the program where you want execution to be suspended. As discussed previously, you can identify locations in a COBOL program using program unit and paragraph names, in addition to source line numbers. When program execution is suspended, you can resume execution by entering the `continue` command.

A program executes until one of the following conditions is true:

- It encounters a breakpoint.
- It calls the `DEBUG` or `PROCESS_DEBUG_` procedure.
- It generates a trap.
- It terminates.

You can also use the following Native Inspect commands to incrementally advance program execution:

- `step`: This command advances program execution one verb at a time. Execution steps into any `PERFORM` or program unit invocations executed within the step range.
- `next`: This command advances program execution one verb at a time. Execution steps over any `PERFORM` or program unit invocations executed within the step range.
- `finish`: This command executes the current process until execution either returns from the current program unit or encounters a debugging event.

The following considerations for execution control might apply:

- Stepping behavior depends on compiler code generation and can vary slightly with different compiler versions.
- You cannot use the `finish` command to step execution out of a `PERFORM`. To step out of a `PERFORM`, you can set a breakpoint at the return location and then continue execution until that breakpoint is encountered.

4 Native Inspect Command Syntax

Categories of Native Inspect Commands

In Table 11, Native Inspect commands are grouped as follows:

- Command Line Options, such as: `help` and `nocstm`.
- Utility Commands, such as: `files` and `dir`.
- Session Control Commands, such as: `attach` (`detach`) and `switch`.
- Snapshot Commands, such as: `save` and `snapshot`.
- Process Control Commands, such as: `continue` and `next`.
- Execution Control Commands, such as: `ih` and `mh`.
- Breakpoint Commands such as: `break` (`tbreak`), and `enable` (`disable`).
- Display and Modify Commands, such as: `break`(`tbreak`), and `enable`(`disable`).
- Stack Commands, such as: `down`and `up`.
- Object and Symbol File Commands, such as: `symbol`, and `ptype`.
- Memory Management Command, where the sole command is `vq`.
- Memory Leak Detection Commands, such as: `info heap-check`, `set heap-check` and `show heap-check`, with their various options.

Table 11 Native Inspect Command Functions

Group Name and Command Name	Function
Command-Line Options	
help Command, help Option	Displays the syntax of all the command-line options.
nocstm Option	Specifies that Native Inspect is not to process the `EINSCSTM` custom file.
version Option	Displays version information about GDB, Tcl, and Native Inspect.
Utility Commands	
amap Command	Displays attributes associated with an address.
base Command	Sets the base for numeric input and output.
cd Command	Changes the current working directory.
comment (#) Command, # (comment) Command	Introduces a comment line.
dir Command	Modifies the search path for source files.
eq Command	Evaluates an expression and displays the result in several bases.
fc Command	Redisplays a previous command for editing and reexecution.
files (ls) Command, ls (files) Command	Displays the files in the current working directory.
fopen Command	Displays information about files that have been opened by the current program.
help Command, help Option	Displays help information about Native Inspect commands.
info Command	Displays information about the debugging target.
log Command	Turns session logging on or off.

Table 11 Native Inspect Command Functions *(continued)*

Group Name and Command Name	Function
map-source-name (map) Command	Defines mapping rules between the source file names at compilation time and at debug time.
pwd Command	Prints the current working directory.
quit (exit) Command, exit (quit) Command	Ends the Native Inspect session.
set Command (environment)	Sets environment settings for Native Inspect.
show Command	Displays environment settings for Native Inspect.
source Command	Reads commands from a file.
version Option	Displays the version of Native Inspect and Tcl.
which Command	Prints file, function, and line information for the specified (text or data) symbol.
Session Control Commands	
attach Command	Associates Native Inspect with the specified process.
detach Command	Disassociates Native Inspect from the current process or from a specified process.
priv Command	Controls the privilege level of the working session (super ID user only).
switch Command	Transfers the current process to Visual Inspect or to Inspect.
vector Command	Changes the process designated as the current process.
wait Command	Waits for the next Debug event or for a **Break** key event.
Snapshot Commands	
save Command	Creates a snapshot file (also known as a save or saveabend file) of the current TNS/E or TNS emulated process.
snapshot Command	Opens a TNS/E native process snapshot file for analysis.
Process Control Commands	
continue Command	Continues the execution of the current process.
finish Command	Executes the current process until execution returns from the currently selected frame.
hold Command	Suspends the current process so that you can perform debugging operations.
jump Command	Resumes execution at the specified location.
kill Command	Terminates the current process.
next (nexti) Command	Advances program execution by one or more statements but steps over any function calls.
step (stepi) Command	Advances program execution by one or more statements, stepping into any called functions.
until Command	Continues execution of the current process until a specified location is reached or until the current stack frame returns.
Execution Control Commands	
mh Command (modify handler)	Modifies signal handlers for a specified signal.
ih Command (info handler)	Displays information about signal handlers.
Breakpoint Commands	
break (tbreak) Command	Sets a code breakpoint (temporarily, in the case of `tbreak`) at a specified line, function, or address.

Table 11 Native Inspect Command Functions *(continued)*

Group Name and Command Name	Function
catch Command	Sets a logical breakpoint on a specified event.
commands Command	Specifies commands that Native Inspect is to execute when a specified breakpoint is hit.
condition Command	Specifies a conditional expression that Native Inspect is to evaluate when a specific breakpoint is hit.
delete Command	Deletes code breakpoints.
disable Command	Disables specified breakpoints.
dmab Command	Deletes a memory access breakpoint (MAB).
enable Command	Enables breakpoints that have been disabled.
ignore Command	Sets the number of breakpoint hits you want Native Inspect to ignore before reporting a specified breakpoint.
mab Command	Sets a memory access breakpoint (MAB).
Display and Modify Commands	
a (an) Command	Displays memory in ASCII format.
dn Command	Displays memory in a specified format.
disassemble (da) Command	Displays a range of memory as instructions.
delete display Command	Deletes an expression from the automatic display list.
disable display Command	Disables automatic display items.
display Command	Adds an expression to the automatic display list.
enable display Command	Enables automatic display items that have been disabled.
env Command	Displays environment information about a process.
fn Command	Searches for a value (finds a number) in the virtual address space of the current process.
in Command	Displays memory as instructions.
modify (mn) Command, info Command with the registers option	Changes the content of memory.
output Command	Displays the value of a specified expression without saving it to the value history list.
print Command	Evaluates and displays the value of a specified expression, saving the result on the value history list.
reg Command	Displays registers.
set Command (variable)	Evaluates an expression and assigns the resulting value to a variable.
x Command	Examines memory at a specified address.
Stack Commands	
bt (tn) Command, tn (bt) Command	Prints a backtrace of all the stack frames.
down (down-silently) Command	Selects the stack frame that is called by the currently selected stack frame.
frame (select-frame) Command	Selects a specified stack frame.
info Command with frame option	Displays information about frames and registers.

Table 11 Native Inspect Command Functions *(continued)*

Group Name and Command Name	Function
tj Command, tu Command	Traces the stack from a TNS/E native jump buffer (`tj` command) or a `ucontext` buffer (`tu` command) contained at the specified address.
up (up-silently) Command	Selects the stack frame that called the currently selected stack frame.
Object and Symbol File Commands	
add-symbol-file Command	Add additional symbol file information.
list Command	Lists source code.
ptype Command	Prints detailed information about a specified data type.
symbol-file (symbol) Command	Opens a TNS/E native code file to build up internal symbol tables.
unload-symbol-file Command	Discards all the symbol data associated with a specified file name.
whatis Command	Displays the data type of a specified expression.
Memory Management Command	
vq Command	Displays information about the extended segments allocated by the current process or changes the currently viewed selectable segment.
Memory Leak Detection Commands	
info Command (memory leak detection)	Displays information about the `corruption`, `heap`, and `leaks` options of the `info` command, used to view memory problems.
set heap-check Command (memory leak detection)	Displays information about the `set heap-check` command and its attributes, used to debug memory problems.
show Command	Shows the `heap-check` option, used to view memory problems.
User-Defined Commands	
define Command	The `define` command defines a command, *`commandname`*, specified by the user.
document Command	Documents the user-defined command, *`commandname`*, so that it can be accessed by help.
show user (see show Command)	Displays definitions (but not documentation) of user-defined commands.

Syntax of Common Command Elements

This section describes the following:

- Syntax of locspec
- Syntax of native-address
- Syntax of llce (low-level conditional expression)
- Syntax of expression
- Syntax of /format

Syntax of locspec

NOTE: *`locspec`* is sometimes referred to as *`linespec`* in files and documents that are used by or related to Native Inspect.

`locspec`

Use *`locspec`* to do the following:

- Specify a single source line with the `list` command to display source lines.
- Specify where to set a code breakpoint with the `breakpoint` command.

You can specify *`locspec`* using any of the following forms:

`line-number`

Specifies a line number in the current file.

`filename:line-number`

Specifies a line number in the specified source file *`filename`*.

`function`

Specifies the line at which the body of the specified function begins.

`filename:function`

Specifies the line at which the body of the specified function begins in the given file *`filename`*. Note that you need to use this option only to avoid ambiguity when identically named functions are in different source files.

`*address`

Specifies the line containing the specified program address. *`address`* can be any decimal or hexadecimal expression; the format is assumed to be decimal by default.

Specifying Code Locations for pTAL Programs

When debugging a pTAL program, you can also use the following notation to specify a code location within a subprocedure:

```
procedure[.subprocedure] ...
```

Specifying Code Locations for COBOL Programs

When debugging a COBOL program, you can also identify a code location using paragraphs, line numbers, program units (including nested program units), instructions, and sections. To specify one of these code locations, use one of the following notations in the line specification:

```
program-unit[.program-unit] ...
[section.]paragraph
paragraph [[in\of] section]
```

These syntax elements enable you to qualify the code location in cases where the code location is not unique within the program.

eal

NOTE:

- In a paragraph number, the leading zero is significant.
- If *`locspec`* consists entirely of digits, and that string of digits matches both a paragraph name and a line number in the program unit, Native Inspect uses the paragraph name. To specify the line number in this case, use the *`filename`*`:`*`line-number`* notation.
- COBOL paragraph names need not be unique within a program unit. To differentiate among paragraphs with the same name, specify a qualified paragraph name using the *`section.paragraph`* or *`paragraph`* `[[in\of]` *`section`*`]` notation. If you specify a paragraph name that is not unique within a program unit, Native Inspect issues an error indicating that the name is ambiguous and that further qualification is required.
- Native inspect does not support the *`program-unit.label`*, or *`program-unit.line-number`* notation.
- Native Inspect does not support setting breakpoints on statement ordinals.

Syntax of native-address

`native-address`

A 32-bit or 64-bit address. You can specify a *`native-address`* by using the following formats:

- Hexadecimal (for example, `0x120001DC0`)
- Decimal (for example, `48331845824`)
- Octal (for example, `044000016700`)

If the address is in the range of `0` through `0xFFFFFFFF`, it is sign-extended to form a 64-bit address.

Example 1 Examples of native-address

Consider the following examples:

- `0x80000` & `0x72000000`

  If the address is greater than `0xFFFFFFFF`, it is treated as a 64-bit address, and no sign extension is done.

- `0xFFFFC5000000`

  The output format for *`native-address`* is `0x`*nnnnnnnnnnnnnnnn* with leading zeros included.

Syntax of llce

`llce`

A low-level conditional expression, used for setting conditional breakpoints with the `break`(see break (tbreak) Command (page 67)) and `mab` command (see mab Command (page 97)).

Low-level conditional expressions are evaluated by the operating system rather than the debugger, yielding faster conditional breakpoint performance. Their capabilities are more limited, however, than those of high-level conditions, which you can use with the `condition` command (seecondition Command (page 71)).

The syntax is:

`-e` *`native-address`*`[&` *`mask`*`]` *`operator value`*

Where:

mask

A 64-bit mask that will be ANDed with the contents of *native-address* before the test is performed with *value*.

operator

The operator is one of the following strings:

- `!=`
- `==`
- `<`
- `>`

value

An integer.

Syntax of expression

expression

A list of operands and operators which, when evaluated, result in a number or a string. Valid operators are those accepted by the source language in which the target being debugged is compiled.

Native Inspect does not support expressions in pTAL. To specify pTAL expressions, use C-style syntax:

pTAL	C-Style	COBOL
`x := y;`	`x = y;`	`AX = A1/A2 + A3 * A4`
`flag := (a<>b);`	`flag = (a != b);`	`FLAG = (A NOT EQUAL TO B)`
pTAL pointer example:	Using C-style pointers:	

`int x[0:9];`
`int .xptr;`
`...`
`@xptr := @x[0];`
`...`
	To display the address, enter:
	`p xptr`	

	To dereference `ptr` (display what `xptr` points to), enter: `p *xptr`	
Variable names are not case-sensitive. (To make Native Inspect recognize pTAL variable names, use the set Command (environment) with the `language ptal` option.)	Variable names are case-sensitive.	Variable names are not case-sensitive.

For additional COBOL considerations, see Evaluating Expressions (page 53).

Other valid expressions are:

$, $$

refers to the last two values printed. For example, `print $$` redisplays the next-to-last value printed.

$*number*

refers to results of previous print commands, which are saved on the value history list. For example, `print $1` redisplays the previous value printed (the output associated with $1), and `print $5` redisplays the output associated with $5.

`$register-name`

displays the contents of the specified register. For example, `print $pc` displays the contents of the `$pc` register.

Syntax of /format

`/format`

A repeat count, followed by a format letter and a size letter in any order as follows:

`/[`*`count`*`][`*`format`*`][`*`size`*`]`

`count`

An integer specifying the number of units of *`size`* to display or print.

`format`

Specifies the format to use for the display, as follows:

Format Specification	Meaning
`o`	octal
`x`	hexadecimal
`d`	decimal
`u`	unsigned decimal
`t`	binary
`f`	float
`a`	address
`i`	instruction (ICODE)
`c`	char
`s`	null terminated string

`size`

Specifies the unit size as follows:

Size Specification	Meaning
`b`	byte
`h`	half word (16 bits)
`w`	word (32 bits)
`g`	giant (64 bits)

For example, `10bx` means 10 repetitions of one byte in hexadecimal.

The *`/format`* options are used in the following commands:

- `x` command (examine), see x Command (page 125).
- `print` command, see print Command (page 103),
- `output` command, see output Command (page 103).

The print and output commands do not accept the following format specifications:

- `i`
- `s`

- Any size letter.
- A repeat count of more than 1.

Specifying Pathnames in Native Inspect Commands

Certain Native Inspect commands require you to specify either an OSS pathname or Guardian file name. Here are the rules Native Inspect follows for resolving pathnames and determining the current working directory.

Resolving Pathnames

If you specify an OSS or Guardian absolute pathname, that pathname is used regardless of whether the current working directory is an OSS or a Guardian working directory.

If you specify an OSS or Guardian relative pathname, it is resolved to an absolute pathname according to the current working directory. Thus, if the current working directory is an OSS absolute pathname, Native Inspect resolves the relative pathname to an OSS absolute pathname. Likewise, if the current working directory is a Guardian absolute pathname, Native Inspect resolves the relative pathname to a Guardian absolute pathname.

Identifying the Default Current Working Directory

The default current working directory is as follows:

- When you start Native Inspect from either the OSS environment (by using the `gtacl -p eInspect` command) or from the Guardian environment (by using the `eInspect` command at the TACL prompt), the current working directory has the Guardian format.
- When a process is placed under Native Inspect control, the current working directory has the OSS format for an OSS process, and the Guardian format for a Guardian process. In normal operations, a process is placed under Native Inspect control when:
  - You enter a `RUND` or `RUNV` command from the Guardian TACL prompt
  - You enter a `run -debug` command from the OSS shell.
  - You enter an `attach` command from the Native Inspect command prompt. See attach Command (page 66).
  - The process calls the `DEBUG()` procedure.
- Opening a snapshot file has no affect on the current working directory, regardless of whether the process in the snapshot file is an OSS process or a Guardian process.
- Commands displaying output containing pathnames are not affect by the current working directory. These commands always display the format returned by the particular command.
- You can switch the current working directory from OSS format to Guardian format, and vice versa, by using the cd Command.

# (comment) Command

An alias for the comment (#) Command, used to embed a comment in a Native Inspect command line.

a (an) Command

A Debug-compatible Tcl command that displays memory in ASCII format.

```
a native-address [count]
```

Where:

native-address

This is the address in memory that you want to display in ASCII. See Syntax of native-address (page 61).

count

This is the amount of memory to display. The default value is one byte.

add-symbol-file Command

Reads debugging symbols from the specified loadfile (file code 800). This command enables you to optionally specify the base address relative to which symbol addresses are computed. The new symbol data is added to the existing data. For more information, see Optionally Loading Symbols Information (page 34).

Related Commands

The `symbol-file` command (see symbol-file (symbol) Command (page 119)) allows you to load a symbol file, but does not allow you to specify the address at which to load symbol.

`add-symbol-file [-g] [-readnow]` *`pathname`* `[`*`address`*`]`

Where:

`-g`

Loads a symbol file with global scope so that symbols are visible to all processes being debugged.

Entered without the `-g` option, loads a symbol file with per-process scope so that symbols are visible only to the current process.

If there is no current process, the added symbol file has global scope, and the `-g` option is optional.

`-readnow`

Expands the symbol table immediately rather than incrementally as needed.

pathname

The OSS pathname or Guardian file name of the TNS/E native code file (with file code 800) that Native Inspect is to open and from which Native Inspect will load symbols.

address

The load address relative to which symbols should be rebased. This is optional for DLLs only. See Debugging DLLs (page 28) for more information

The `add-symbol-file` command prompts you before reading in the loadfile's symbols regardless of whether you include an address argument. The prompt displays the address at which Native Inspect will base the symbols. If you reply `No`, the command is aborted. If you reply `Yes`, the symbols are read in.

amap Command

Displays attributes associated with a specified address.

`amap` *`address`*

Examples

```
(eInspect 4,1010): amap 0x8000290
pcKind = 19
(eInspect 4,1010): amap 0x70000000
pcKind = 8 ReadOnly Code ElfText PrivToWrite
```

attach Command

Associates Native Inspect with a specified process that must be executing in the same CPU as the Native Inspect process. The `attach` command enables you to debug a running process using Native Inspect.

```
attach [pin] | [$process-name]
```

Where:

`pin`

The process ID (process number) of the process you want to attach. The command fails if you specify an invalid or nonexistent *`pin`*. To attach to a running OSS process, you must use its Guardian process ID.

`$`*`process-name`*

The name of the process or process-pair you want to attach. The command fails if you specify an invalid or nonexistent process name. For process-pairs, Native Inspect attaches to either the primary or backup process of a process-pair depending on the CPU in which the Native Inspect process is being executed. To attach to the primary process of a process-pair, Native Inspect must be running in the same CPU as the primary process. If the backup process is executing in the same CPU as the Native Inspect process, Native Inspect prompts you for confirmation before attaching to the backup process.

The `attach` command issues a `PROCESS_DEBUG_` request for the specified process, and Native Inspect then waits for the next debugging event. (When privileged debugging is enabled, the attach command specifies the `DEBUGNOW` option.)

Then when the process enters debug, Native Inspect receives the debugging event, creates a session for the process (adding the process to the set of processes being debugged), and makes the process the current process.

If a debugging event occurs for another process when Native Inspect is waiting (that is, between the time you enter the `attach` command and when the specified process enters debug), that other process then becomes the current process.

Example

See Example of Using Multiple Instances of Native Inspect (page 26).

base Command

A Debug-compatible Tcl command that sets the default base for numeric input and output.

The `base` command is an alias for the `set` command with the `radix`, `input-radix`, and `output-radix` options. See set Command (environment)

```
base {input|output} {HEX|OCT|DEC}
```

Where:

`input`

Sets the base used for numeric input.

`output`

Sets the base used for numeric output.

`HEX`

Sets the base in hexadecimal.

`OCT`

sets the base in octal.

`DEC`

Sets the base in decimal.

break (tbreak) Command

The `break` command sets an instruction breakpoint for the current process, at a specified line, function, or address. The `tbreak` command is similar, but sets a temporary breakpoint that is deleted after it is hit.

Related Commands: Use the `enable` command (see enable Command (page 78)) and `disable` command (see disable Command (page 75)) to enable and disable breakpoints, respectively.

```
{break|tbreak} [locspec] [flags] [-e llce | if cond-exp]
```

Where:

`locspec`
: The location at which you want to set a breakpoint. See Syntax of locspec (page 59).

`flags`
: Either or both of the following:

  `-g`
  : Indicates a global breakpoint, which can be set only by the super ID user after issuing the `priv` command. See priv Command (page 108).

  `-h`
  : Indicates a halt loop breakpoint, which can be set only by the super ID user after issuing the `priv` command. See priv Command (page 108).

`llce`
: A low-level conditional expression (cannot be specified with the `-h` flag). See Syntax of llce (page 61).

`cond-exp`
: A language expression that produces an integer or Boolean result.

For COBOL-specific considerations, see Evaluating Expressions (page 53).

Setting Conditional Breakpoints

You can set a conditional breakpoint with the `break` or `tbreak` command by including either an *`llce`* (a low-level conditional expression, which is evaluated by the breakpoint interrupt handler) or a source language expression (evaluated by the debugger). In either case, the breakpoint is reported only if the expression evaluates to True or a nonzero result. Otherwise, execution of the program continues. For information about specifying *`llce`*, see Syntax of llce (page 61).

You can also use the condition Command to associate a condition with a previously set breakpoint.

Setting Global Breakpoints

Global breakpoints are breakpoints that are set using the `-g` option. Setting global breakpoints has the following constraints:

- You must be logged on as the super ID.
- You must turn on privileged mode by using the `priv` command. (See priv Command (page 108).)
- The operation is exclusive. No other user can perform global debugging at the same time.

A global breakpoint is triggered by any process that executes the code on which the global breakpoint is set.

NOTE: Set global breakpoints when debugging a problem in code that is shared by multiple processes. However, use global breakpoints with care because they might result in numerous processes being suspended in the debugger.

Examples

- To set code breakpoints:

```
(eInspect 1,325): b 216
Breakpoint 1 at 0x70001670:2: file \SIERRA.$YOSE1.SYMBAT1.SCXXTST, line 216.
(eInspect 1,325): b test_complexTypes
Breakpoint 2 at 0x70003700:0: file \SIERRA.$YOSE1.SYMBAT1.SCXXTST, line 420.
```

- To set a code breakpoint in a COBOL nested program unit:

```
(eInspect 4,668): b main.main-level2
Breakpoint 1 at 0x70002280:2: file \SIERRA.$DIVA.CBDEMO, line 41.
```

- To set a breakpoint on a qualified paragraph in a COBOL program:

```
(eInspect 4,751): b first-para of last-section
Breakpoint 1 at 0x70004500:2: file \SIERRA.$DIVA.CBDEMO, line 41.
(eInspect 2,640):b Para1
Breakpoint 2 at 0x70003be0:0: file \SIERRA.$DIVA.CBEX.CBEXAM1, line 75.
(eInspect 2,640): b Para2 OF Sec2
Breakpoint 3 at 0x70004440:0: file \SIERRA.$DIVA.CBEX.CBEXAM1, line 97.
(eInspect 2,640): b Sec3.Para2
Breakpoint 4 at 0x70004a20:0: file \SIERRA.$DIVA.CBEX.CBEXAM1, line 109.
```

- To list breakpoints:

```
(eInspect 1,325): info break
Num Type           Disp Enb Address     What
1   breakpoint     keep y   0x70001672  in main
                                        at \SIERRA.$YOSE1.SYMBAT1.SCXXTST:216
 2   breakpoint     keep y   0x70003700 in test_complexTypes
                                        at \SIERRA.$YOSE1.SYMBAT1.SCXXTST:420
```

For COBOL programs, the `info breakpoints` command displays the paragraph and section names along with the module and file information, if the breakpoint is placed on the paragraph or section.

```
(eInspect 2,640):info break
Num Type           Disp Enb Glb Address    What
2   breakpoint     keep y   n   0x70003be0 at PARA1 in MAIN-PROGRAM
                                              at \SIERRA.$DIVA.CBEX.CBEXAM1:75
3   breakpoint     keep y   n   0x70004440 at PARA2 OF SEC2 in MAIN-PROGRAM
                                              at \SIERRA.$DIVA.CBEX.CBEXAM1:97
4   breakpoint     keep y   n   0x70004a20 at PARA2 OF SEC3 in MAIN-PROGRAM
                                              at \SIERRA.$DIVA.CBEX.CBEXAM1:109
```

- To define and list conditional breakpoint:

```
(eInspect 1,329): b 352
Breakpoint 1 at 0x70002540:0: file \SIERRA.$YOSE1.SYMBAT1.SCXXTST, line 352.
(eInspect 1,329): condition 1 pcb->pin == 2
(eInspect 1,329): info break
Num Type Disp Enb Address What
1 breakpoint keep y 0x70002540 in pcbDataStructs_initialize
at \SIERRA.$YOSE1.SYMBAT1.SCXXTST:352
stop only if pcb->pin == 2
```

- To set a breakpoint at a code address:

```
(eInspect 3,663):b *0x70002c40:2
Breakpoint 3 at 0x70002c40:2: file \SIERRA.$YOSE1.SYMBAT1.SCXXTST, line 372.
(eInspect 3,663):c
Continuing.

Breakpoint 3, 0x70002c40:2 in pcbDataStructs_initialize ()
    at \SIERRA.$YOSE1.SYMBAT1.SCXXTST:372
  372         PCB_addAttribute( pcb, PCBAttribute_createNonstop( PCBList.entry[11] ) );
```

- To define and list a MAB:

```
(eInspect 0,294): mab globStr.f2 -c
Memory access breakpoint 2 (mab)
(eInspect 0,294): info break
```

```
Num Type Disp Enb Glb Address What
2 mem access brk keep y n globStr.f2 -c
```

bt (tn) Command

Prints a backtrace of all stack frames. Frame numbers are preceded by a number sign, (#).

Alias: `tn`.

`{bt|tn} [`*`count`*`]`

Where:

`count`

An integer that displays a backtrace of the innermost *`count`* frames. If you specify a negative count, then a backtrace of the outermost -*`count`* frames is displayed.

Consideration for Debugging TNS Processes

Although Native Inspect does not debug TNS processes, the `bt` command displays a TNS stack trace.

Example

To display the current frame:

```
(eInspect 4,770):bt
 #0  pcbDataStructs_initialize () at \SIERRA.$YOSE1.SYMBAT1.SCXXTST:362
 #1  0x700016a0:0 in main (argc=1, argv=0x8003010)
     at \SIERRA.$YOSE1.SYMBAT1.SCXXTST:217
 #2  0x700011f0:0 in _MAIN () at \SPEEDY.$RLSE.T8432H01.CPLMAINC:68
```

catch Command

Sets a logical breakpoint on a specified event (a catch event). A catch event is similar to a breakpoint except that a catch event is associated with a logical event rather than a code location.

`catch `*`event`*

Where:

`event`

The event on which the process will be held. Supported events are:

- `STOP` – Holds the process on a stop event.
- `ABEND` – Holds the process on an abend event.
- `LOAD [`*`dllname`*`]` – Holds the process on the loading of a DLL. If you specify *`dllname`*, the process is held when the specified DLL is loaded. If you omit *`dllname`*, the process is held when any DLL is loaded.
- `UNLOAD [`*`dllname`*`]` – Holds the process on the unloading of a DLL. If you specify *`dllname`*, the process is held when the specified DLL is unloaded. If you omit *`dllname`*, the process is held when any DLL is unloaded.

Catch events are treated as breakpoints and each catch event has an associated breakpoint number. You can use all the breakpoint-related commands to manage catch events. This includes the following:

- `commands` (see commands Command (page 70)).
- `condition` (see condition Command (page 71)).
- `delete` (see delete Command (page 73)).
- `enable` (see enable Command (page 78)).

- `disable` (see disable Command (page 75)).
- `info` with the `breakpoints` option (see info Command (page 86)).

To display a list of current catch events, use the `info` command (see info Command (page 86)) with the `breakpoints` option.

Managing a Stopping Process (STOP and ABEND Events)

When a process triggers a `STOP` or `ABEND` event, the process is in a stopping state. You can examine a process that is in a stopping state, but the process cannot execute any further. Execution control commands are disabled when a process is suspended at a `STOP` or `ABEND` event. You can use the `save` command to create a snapshot of a stopping process. See save Command (page 109).

You can switch a stopping process to another debugger by using the `switch` command. The process is eventually given back to the original instance of Native Inspect, and the process will still be in the stopping state. See switch Command (page 119).

After you examine a stopping process, you must use the `continue`, `detach`, or `kill` command to disassociate the process from Native Inspect and allow the process to terminate.

cd Command

Changes the current working directory to the specified pathname.

Alias: `volume`.

`cd` *`pathname`*

Where:

`pathname`

The OSS or Guardian absolute or relative pathname of the new current working directory.

Example

- To specify that the current working directory (Guardian environment) is `mysubvol` on the $DATA5 disk volume:

  ```
  (eInspect 1,325):cd $data5.mysubvol
  ```

- To specify that the current working directory (OSS environment) is `/usr/mysrc`:

  ```
  (eInspect 1,1032):cd /usr/mysrc
  ```

commands Command

Sets the commands to be executed when a breakpoint or catch is hit. Enter the commands starting on the next line.

`commands [`*`breakpoint-number`*`]`

Where:

`breakpoint-number`

The number of the breakpoint, catch event, or MAB for which the specified expression is to be evaluated.

Example

To define breakpoint actions:

```
(eInspect 1,329): commands 1
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just "end".
>print PCB
>end
```

comment (#) Command

A Tcl command that introduces a line of comment.

Alias: # command. (See # (comment) Command (page 64)).

`{comment|#} [`*`text`*`]`

Either `comment` or # must be the first non-blank character in the line. The entire line is then treated as a comment.

condition Command

Specifies a conditional expression to be evaluated when a breakpoint is hit. The breakpoint is reported only if the condition evaluates to `TRUE`.

`condition `*`breakpoint-number`*` [`*`conditional-expression`*`]`

Where:

`breakpoint-number`,

The number of the breakpoint or catch event, for which the specified expression is to be evaluated. You canconditionalize MABs in the same manner as breakpoints by using the MAB's ordinal as an argument.

`conditional-expression`,

The conditional expression that you want evaluated when Native Inspect encounters the specified breakpoint. If you omit the *`conditional-expression`*, any existing condition is cleared, and the specified breakpoint is treated as an unconditional breakpoint.

The conditional expression here is different from the low-level conditional expression supported by the `mab` and `break` commands. Low-level conditional expressions (`llce`) are evaluated by interrupt processes in the NonStop operating system. High-level conditional expressions (HLCEs), such as those supported by the `condition` command, are evaluated by Native Inspect.

continue Command

Continues execution of the current process. Native Inspect suspends command prompting until the next debugging event occurs, or until you press the **Break** key.

`continue [`*`ignore-count`*`]`

Where:

`ignore-count`

Specifies the number of times to ignore a breakpoint at the current location.

Example

To continue execution:

```
(eInspect 1,329):c
Continuing.
Breakpoint 2, pcbDataStructs_initialize ()
    at \SIERRA.$YOSE1.SYMBAT1.SCXXTST:386
  386                  pcb = PCBList.entry[2]->ref.pcb;
```

define Command

The `define` command defines a command, *`commandname`*, specified by the user. If there is already a command by that name, you are asked to confirm that you want to redefine it.

The definition of the command is made up of other Native Inspect command lines, which are given following the `define` command. The end of these commands is marked by a line containing `end`.

`define `*`commandname`*

Where:

commandname

The name of the command to be defined. If a command by that name already exists, you are asked to confirm if you want to redefine that command.

Usage Note

A user-defined command is a sequence of commands to which you assign a new name as a command. This is done with the `define` command. User commands can accept up to 10 arguments separated by whitespace. You access arguments within the user-defined command by specifying `$arg0` ... `$arg9`.

Example

- This defines the command `adder`, which prints the sum of its three arguments. The arguments are text substitutions, so they may reference variables, use complex expressions, or even perform further functions calls.

  ```
  define adder
    print $arg0 + $arg1 + $arg2
  end
  ```

  To execute the command `adder`, use:

  ```
  adder 1 2 3
  ```

- This is an example of defining the user-defined command `xyz`. This gives you a new command to print the value of the variable `xyz` in hex (/x).

  ```
  (eInspect 0,144):define xyz
  Type commands for definition of "xyz".
  End with a line saying just "end".
  >p /x xyz
  >end
  ```

dn Command

A Debug-compatible Tcl command that displays memory in the format you specify.

`dn` *`native-address`* `[`*`count`*`] [:`*`format`*`]`

Where:

native-address

The address at which you want to display memory. The `d` command accepts only 32-bit and 64-bit addresses. See Syntax of native-address (page 61).

count

The number of items to display. The default value is one.

:*format*

The format in which to display memory. Options are:

- *a* for ASCII
- *I* for ICODE (instruction code)
- `o [`*`n`*`]` for octal.
- `d [`*`n`*`]` for decimal.
- `h [`*`n`*`]` for hexadecimal.

Where `[`*`n`*`]` is the bit size, expressed as 8, 16, 32, or 64. The default value is 32.

If you omit the *format* option, 32-bit quantities are displayed in the default output base.

Example

To display memory using the Debug-compatible a and d commands:

```
(eInspect 3,663): d 0x8005884 10
0x8005884:     0x6d6f6e69     0x746f7200     0x00000000     0x00000000
0x8005894:     0x00000000     0x00000000     0x00000000     0x00000000
0x80058a4:     0x00000000     0x00000000

(eInspect 3,663): a 0x8005884 10
0x8005884:  .moni.  .tor..  ......  ......
0x8005894:  ......  ......  ......  ......
0x80058a4:  ......  ......
```

delete Command

Deletes code breakpoints.

If you not specify any breakpoint numbers, Native Inspect prompts you for confirmation before deleting all the breakpoints associated with the current process.

`delete [breakpoints][`*`breakpoint-number`*`...]`

Where:

`breakpoint-number`

The number of the breakpoint or catch event that you want deleted. You can delete only breakpoints that are associated with the current process or global breakpoints. Deleted breakpoint numbers are not reassigned.

To determine a breakpoint number, use the info Command with the `breakpoints` option.

delete display Command

Deletes an expression from the list of expressions that are automatically displayed when the program is suspended.

If you do not specify a value for *`number`*, the command deletes all automatic display expressions.

`delete display [`*`number`*`]`

Where:

`number`

An ordinal number that identifies one expression on the automatic display list.

Consideration

- To specify an expression for automatic display, use the `display` command. (See display Command (page 76).)
- To display the expressions on the automatic display list do the following:
  - Use the `info` command with the `display` option. (See info Command (page 86).)
  - Use the `display` command with no arguments.

detach Command

Disassociates Native Inspect from the current process or from a specified process. Removes the current process from the set of processes being debugged and allows the process to continue executing.

If only one process is being debugged by Native Inspect, there is no current process after you enter the detach command.

If you do not enter a pin or process name, this command detaches Native Inspect from the current process.

Related Commands: Detach is the complement of the `attach` command. (See attach Command (page 66).)

`detach [`*`pin`*`] | [$`*`process-name`*`]`

Where:

`pin`

The process number of a process under the control of Native Inspect from which you want to detach.

`$`*`process-name`*

The name of the process or process-pair under the control of Native Inspect from which you want to detach.

Considerations

- Before entering the detach command, you should typically clear all breakpoints in the process and continue process execution. If breakpoints are set in the process, Native Inspect displays a confirmation prompt.
- If the process you name in a `detach` command is suspended, it will automatically be resumed.
- If you enter the `detach` command when breakpoints are set in the process, Native Inspect issues a warning describing the situation and gives you the option of continuing or aborting the operation.

Example

See the section titled: Example of Using Multiple Instances of Native Inspect (page 26)

dir Command

Appends a specified directory to the search path that Native Inspect uses to locate source files. The directory search path is shared by all programs that are currently being debugged by Native Inspect.

You typically use the `dir` command to help Native Inspect find source files when their current location differs from the location at which they were compiled, but their base file name remains the same. You use the related `map` command (see map-source-name (map) Command (page 99)) to specify base file name changes between compilation and debugging.

If you do not specify a directory, the search path is reset to empty (that is, Native Inspect searches for source files only in the directory from which the source file was compiled).

`dir [`*`directory`*`]`

Where:

`directory`

The name of a local NonStop Guardian subvolume (`$volume.subvolume`) or an OSS directory (`/h/usr/rell/src`) that you want to append to the search path for source files.

Related Commands: The `list` command (see list Command (page 94)), the `map` command (see map-source-name (map) Command (page 99)), and the `show` command with the directories option (see show Command (page 116)).

Example

In the following example, `$cdir` represents the compilation directory, and `$cwd` represents the current working directory. To set a source subvolume search path and display the current subvolume search path, use the following commands:

```
(eInspect 5,855): dir $d0117.test
Source directories searched: $d0117.test:$cdir:$cwd
(eInspect 5,855): dir $d0117.kris
```

```
Source directories searched: $d0117.kris:$d0117.test:$cdir:$cwd
(eInspect 5,855): show directories
Source directories searched: $d0117.kris:$d0117.test:$cdir:$cwd
```

disable Command

Disables specified breakpoints, which remain defined but are not hit until reenabled. Use the `enable` command (see enable Command (page 78)) to enable a disabled breakpoint.

If you do not enter any breakpoint numbers, the command disables all the breakpoints associated with the current process.

Abbreviations `dis` and `disa`.

`disable [breakpoints] [`*`breakpoint-number`*`...]`

Where:

`breakpoint-number`

The number of a breakpoint or catch event that you want disabled. If you omit *`breakpoint-number`*, all current breakpoints are disabled.

Example

See the example for the enable Command (page 78).

disable display Command

Disables the evaluation and display of a previously defined automatic display expression.

If you do not specify any command arguments, the command disables the entire automatic display list.

`disable display [`*`number`*`]`

Where:

`number`

An ordinal number that identifies an expression to be deleted from the automatic display list.

Considerations

- To specify an expression for automatic display, use the `display` command (see display Command (page 76)).
- To display the current expression numbers on the automatic display list, either enter the `info` command (see info Command (page 86)) with the `display` option, or enter the `display` command with no arguments.

disassemble (da) Command

Displays a range of memory as instructions. To display the instructions that compose the current line, first use the `info line` command to display the line's address range and then use the `disassemble` command.

NOTE: COBOL programs are often lengthy. For this reason, using the `disassemble` command with a COBOL program can result in a lengthy display of instructions. HP recommends displaying the instructions for one line at a time, as described in the preceding paragraph.

If you specify only one of *`start-address`* and *`end-address`*, Native Inspect disassembles the entire function surrounding the given address.

If you specify no *`start-address`* or *`end-address`*, Native Inspect disassembles the entire function surrounding the current PC value.

Alias: `da`.

```
{da|disassemble} [[start-address][end-address] | function-name]
```

Where:

start-address

Specifies the starting address of the range of instructions to display.

end-address

Specifies the ending address of the range of instructions to display.

function-name

Specifies a function to display.

NOTE: You can use the `x` command to display a specified number of instructions starting at a specified address.

Example

To display instructions for a function:

```
(eInspect 3,663): da PCB_addAttribute
Dump of assembler code for function PCB_addAttribute:
;;; File: \SIERRA.$YOSE1.SYMBAT1.SCXXTST
  239              {
0x70001820:0 <PCB_addAttribute>:        [MII]       alloc r34=ar.pfs,30,30,0
0x70001820:1 <PCB_addAttribute+6>:                  adds r12=-96,r12
0x70001820:2 <PCB_addAttribute+12>:                 nop.i 0x0;;
0x70001830:0 <PCB_addAttribute+16>:     [MMI]       adds r27=32,r12
0x70001830:1 <PCB_addAttribute+22>:                 nop.m 0x0
0x70001830:2 <PCB_addAttribute+28>:                 nop.i 0x0;;
0x70001840:0 <PCB_addAttribute+32>:     [MII]       stf.spill [r27]=f2
0x70001840:1 <PCB_addAttribute+38>:                 mov r35=r32
0x70001840:2 <PCB_addAttribute+44>:                 mov r36=r33
0x70001850:0 <PCB_addAttribute+48>:     [MMI]       adds r37=48,r12;;
0x70001850:1 <PCB_addAttribute+54>:                 st8 [r37]=r35
0x70001850:2 <PCB_addAttribute+60>:                 nop.i 0x0
0x70001860:0 <PCB_addAttribute+64>:     [MMI]       adds r38=56,r12;;
0x70001860:1 <PCB_addAttribute+70>:                 st8 [r38]=r36
0x70001860:2 <PCB_addAttribute+76>:                 nop.i 0x0
  240             pcb->attribute[ pcb->attributeCount++ ] = pcbAttribute;
0x70001870:0 <PCB_addAttribute+80>:     [MMI]       adds r39=60,r12;;
0x70001870:1 <PCB_addAttribute+86>:                 ld4 r40=[r39]
0x70001870:2 <PCB_addAttribute+92>:                 nop.i 0x0
...
```

NOTE: The output offsets are specified in hexadecimal.

display Command

Specifies an expression that is to be automatically evaluated and the result displayed each time the program is suspended.

If you do not specify any arguments, Native Inspect displays the expressions currently on the automatic display list.

```
display [[/format] expression]
```

Where:

/format

An optional count, format, and size specification. See Syntax of /format (page 63).

expression

Evaluates the expression and adds it to the list of expressions to be evaluated. See Syntax of expression (page 62).

Example

- To use the automatic display list:

```
(eInspect 4,798): display pcb->attributeCount
1: pcb->attributeCount = 1
(eInspect 4,798): display pcb->pin
2: pcb->pin = 0
(eInspect 4,798): next
  364                    PCB_addAttribute( pcb, PCBAttribute_createNonstop( PCBLis
t.entry[10] ) );
2: pcb->pin = 0
1: pcb->attributeCount = 2
(eInspect 4,798): next
  365                    memcpy( gBuffer, pcb, sizeof( PCB_t ) );
2: pcb->pin = 0
1: pcb->attributeCount = 3
(eInspect 4,798): disable display 1 2
(eInspect 4,798): next
  369                    pcb = PCBList.entry[0]->ref.pcb;
(eInspect 4,798): info display
Auto-display expressions now in effect:
Num Enb Expression
2:   n  pcb->pin
1:   n  pcb->attributeCount
```

- To display a COBOL expression:

```
(eInspect 7,383): display CI
1:CI=142
(eInspect 7,383): next
00142
 103              DISPLAY “Leaving cat”.
1: CI=142
(eInspect 7,383): display CI NOT EQUAL 0
2: CI NOT EQUAL 0=’T’
```

dmab Command

Deletes a memory access breakpoint (MAB), which is set with the `mab` command (see mab Command (page 97)).

```
dmab [-g]
```

Where:

`-g`
: Deletes the global MAB.

Example

To delete the memory access breakpoint:

```
(eInspect 1,480): dmab
```

document Command

Documents the user-defined command, *`commandname`*, so that it can be accessed by help. The command *`commandname`* must already be defined. The `document` command reads lines of documentation, just as the `define` command reads the lines of the command definition, ending with `end`.

After the `document` command is finished, running `help` on command *`commandname`* displays the documentation you have written.

To change the documentation of a command, use the `document` command again. Redefining the command with `define` does not change the documentation.

`document` *`commandname`*

Where:

commandname

The name of the command to be defined. If a command by that name already exists, you are asked to confirm if you want to redefine that command.

Usage Note

A user-defined command is a sequence of commands to which you assign a new name as a command. This is done with the define Command (page 71). User commands can accept up to 10 arguments separated by whitespace. You access arguments within the user-defined command by specifying `$arg0` ... `$arg9`.

down (down-silently) Command

Selects the stack frame that is called by the current stack frame. The selected stack frame becomes the stack frame relative to which program state is displayed. The `down` command also prints out information about the selected stack frame.

Related Commands: The `up` command (see up (up-silently) Command (page 123)).

`{down|down-silently} count`

Where:

count

The number of frames to traverse before selecting a frame.

Example

See the Example in the section titled: up (up-silently) Command (page 123).

enable Command

Enables breakpoints that you have disabled using the `disable` command (see disable Command (page 75)).

If you do not enter any breakpoint numbers, the command enables all the breakpoints associated with the current process.

`enable [once|delete] [breakpoint-number...]`

Where:

`once`

Enables the specified breakpoint and then disables it after it is hit once.

`delete`

Enables and then deletes the specified breakpoint after it is hit.

breakpoint-number

The number of a disabled breakpoint or catch event that you want to enable.

Example

To disable and then enable a breakpoint:

```
(eInspect 1,329): disable 1
(eInspect 1,329): info break
Num Type           Disp Enb Address    What
1   breakpoint     keep n   0x70002540 in pcbDataStructs_initialize
                                       at \SIERRA.$YOSE1.SYMBAT1.SCXXTST:352
        stop only if pcb->pin == 2
        print PCB
(eInspect 1,329): enable 1
(eInspect 1,329): info break
Num Type           Disp Enb Address    What
```

```
1   breakpoint     keep y   0x70002540 in pcbDataStructs_initialize
                                       at \SIERRA.$YOSE1.SYMBAT1.SCXXTST:352
        stop only if pcb->pin == 2
        print PCB
```

enable display Command

Enables the evaluation and display of a previously disabled display expression.

If you do not specify a *number*, the command displays all the previously displayed expressions on the automatic display list.

`enable display [number]`

Where:

number
: An ordinal number that identifies an expression on the automatic display list.

Considerations

- To specify an expression for automatic display, use the `display` command (see display Command (page 76)).
- To display the expression numbers on the automatic display list, either use the `info` command (see info Command (page 86)) with the `display` option, or enter the `display` command with no arguments.

env Command

Displays information about the environment of the current process, including register values; segment numbers for data space, code space and user stack; and user segment information.

`env`

Native Inspect also displays the current input and output base, whether logging is currently on or off, and the number of lines in a page (`show height`).

For process debugging, the terminal name and the privileged mode (`show priv`) are also displayed.

eq Command

A Debug-compatible Tcl command that evaluates an expression and displays the result in octal, decimal, hexadecimal, and ASCII.

`eq expression`

Where:

expression
: An expression used in the current process.

exit (quit) Command

An alias for the `quit` command (see quit (exit) Command (page 108)). This command closes the current Native Inspect session but leaves the current process running.

If you enter the `quit` or `exit` command when breakpoints are set in the process or when the process is suspended, Native Inspect issues a warning and gives you the option of continuing or aborting the operation.

fc Command

Redisplays a previous command and allows you to edit and reexecute the command. This command behaves like the `fc` command in TACL on a NonStop system.

```
fc [command-number|command-string]
```

Where:

`command-number`

The number of the command you want to redisplay, edit, and reexecute. Use the `show` command (see show Command (page 116)) with the `commands` option to display the commands entered in the current session.

`command-string`

The first few letters of the command you want to redisplay, edit, and reexecute.

Native Inspect supports the same editing characters that TACL supports:

- `d` or `D` for delete (deletes the characters above the `d` or `D`)
- `i` or `I` for insert (inserts the string that follows the `i` or `I`)
- `r` or `R` for replace (replaces the characters in the original command with the characters following the `r` or `R`)

See the *Guardian User's Guide* for examples of the FC command.

files (ls) Command

Displays files in the current working directory or the directories that match a specified pattern.

If you do not specify a pattern, the `files` command displays the names of all the files in the current working directory.

Alias: `ls` (see ls (files) Command (page 97)).

```
files [pattern]
```

Where:

`pattern`

Any wildcard pattern, such as `?disp*`, which matches the files named:`ODISP`, `ODISP2`, and `ADISPFILE`).

The files command accepts the following wildcard characters:

- `*` – Matches 0 or more characters
- `?` – Matches any one character

finish Command

Executes the current process until execution either returns from the currently selected frame (by default, the current frame) or encounters a debugging event. The `finish` command prints the return value, if there is one. Use the `frame` command (frame (select-frame) Command (page 82)) to select the current frame.

The `finish` command operates relative to the currently selected frame, which might differ from the current execution frame.

After you enter a finish command, you can repeat it by pressing the **Enter** key at the next Native Inspect prompt. The effect is the same as if you had entered another finish command. The ability to repeat continues until you enter any other Native Inspect command.

```
finish
```

Related Commands: `next` (see next (nexti) Command (page 102)), `step` (see step (stepi) Command (page 118)).

fn Command

Searches for a specified value (finds a number) in the virtual address space of the current process. The `fn` command is Debug-compatible.

```
fn value [start-addr [end-addr]] [type]
```

Where:

value
: The value you want to find. The `fn` command does not infer the bit size of *value* based on its magnitude.

start-addr
: The address where the search is to begin.

end-addr
: The address where the search is to end.

type
: The bit size of *value* (8, 16, 32, or 64 bits). The default value is 32 bits. The search increment is the same as the bit size.

After finding the first instance of the specified *value*, Native Inspect prompts you to confirm continuation of the search.

fopen Command

Displays information about files that have been opened by the current program. This command is applicable to TNS/E native processes as well as snapshot files.

```
fopen [file-num] [-d]
```

Where:

file-num
: is the number of the file about which you want information. If you do not specify a file number, Native Inspect displays information about all the files opened by the program. To get information about the last file system error, specify *file_num* as -1.

-d
: is the flag indicating detailed output is requested. The command output includes information such as the file system error, error detail, device type, device subtype, and so on.

Considerations

The `fopen` command does not accept or list COBOL unit descriptors. `fopen` lists and accepts Guardian file numbers, and lists all files when no arguments are specified.

Examples

- To display concise information about all the files opened by the program:

```
    (eInspect 0,56):fopen
FileNum  LastErr  Name
1        0        \PELICAN.$ZTN2.#PT9WNC3
2        0        \PELICAN.$SYSTEM.#0020780
3        0        \PELICAN.$DATA4.TEST1.CPU0
```

- To display detailed information about a file:

```
  (eInspect 0,56):fopen 2 -d
Name     \PELICAN.$SYSTEM.#0020780
Filenum  2
            General File Information.
   Device Type                 3
   Device Subtype              53
   File Type                   UNSTRUCTURED (0)
   Object Type                 0
   Logical Device Number       6
   Open Access                 1
   Open Exclusion              0
```

```
      Open Nowait Depth            0
      Open Sync Depth              0
      Open Options                 0
      Physical Record Length       4096 Bytes
      Outstanding Requests         0
      Error                        0
      Error Detail                 0
               Disk File Information.
      End of File                  0 Bytes
      Current Record Pointer       0
      Next Record Pointer          0
      Modification Timestamp       2008-03-03 15:52:52
      Extent Size                  28 Pages, 28 Pages
      File Code                    0
      Flags                        DEMOUNTABLE WRITE-THRU
      Block length                 0 Bytes
      Logical Record Length        0 Bytes
      Maximum Extents              0
      Partitions                   0

   (eInspect 2,1071):fopen 3 -d
   Name     /home/vi/a
   Filenum  3
               OSS File Information.
      Mode                         32768
      File Descriptor(s)           0
      Error                        0
      Error Detail                 0
               OSS Disk File Information.
      UID                          65535
      GID                          255
      Serial Number                21405
      End of File                  8 Bytes
      Device ID                    541165879296
      RDev                         0
      Access Timestamp             2008-02-21 13:49:37
      Change Timestamp             2008-02-21 13:49:46
      Modification Timestamp       2008-02-21 13:49:46
      Link Count                   1
```

- To display information about last file system error:

```
     (eInspect 0,1136):fopen -1
   FileNum  LastErr  Name
   -1       11
```

frame (select-frame) Command

Selects a stack frame and prints information about the selected stack frame. The selected stack frame becomes the frame relative to which program state is displayed.

The `select-frame` is a silentversion of the frame command, does not print out information about the selected frame.

`[frame|select-frame] [`*`frame-number`*`]`

Where:

`frame-number`

The number of the frame you want to select. To display frame numbers, use the `bt` command (see bt (tn) Command (page 69)). The frame at which execution is currently halted is numbered 0, and frame numbers continue consecutively to the base frame from which execution began.

If you do not include any arguments, the `frame` command displays information about the current stack frame, which can be useful for determining your current program location.

Considerations

The currently selected frame (specified in the frame command) is distinguished from the current program location (the frame at which execution is suspended), subject to the following conditions:

- Most Native Inspect commands operate on the currently selected frame.
- Execution-control commands, such as step and next, operate on the current program location.
- For COBOL programs, the CALL stack records the history of active program unit invocations. By default, Native Inspect shows the program state relative to the most recently invoked program unit.
- You can use the `frame` command to view program state relative to some other frame.

NOTE: PERFORM invocations are not listed on the CALL stack.

Examples

- To display the current frame:

```
(eInspect 6,679): frame
#0  test_complexTypes() () at \SYS04.$D0117.SYMBAT1.SCXXTST:424
424           printf( "%s test_complexTypes\n", getStepPrefix( 1 ) );
```

- To display frame number 1:

```
(eInspect 6,679): frame 1
#1  0x70001570:0 in main (argc=1, argv=0x8003010)
    at \SYS04.$D0117.SYMBAT1.SCXXTST:218
  218                  test_complexTypes();
```

help Command, help Option

Displays information about commands of Native Inspect. The online help for Native Inspect has been inherited from its WDG/GDB parents and thus provides somewhat different information than that provided in this manual.

Command or Option	Function
help command	Takes several arguments and displays syntax, explanation, and examples according to the options you specify.
help command-line option (`--help`)	Takes no arguments and displays the syntax of all the command-line options `nocstm` and `version`. (See nocstm Option and version Option).

`help [`*`command`*`]`

Where:

`help`

entered with no options at the Native Inspect prompt, displays a list of general classes of Native Inspect commands for which you can display help.

`command`

The name of a Native Inspect command for which you want to display help.

Examples

- To display help:

```
(eInspect 3,-2):  help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
```

```
data -- Examining data
filecmd -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in that class.
Type "help" followed by command name for full documentation.
Type "tcl" followed by command name for Tcl commands (e.g., "tcl help").
Command name abbreviations are allowed if unambiguous.
```

- To display help for the symbol command:

```
(eInspect 3,-2): help symbol
Load symbol table from executable file FILE.
The `file' command can also load symbol tables, as well as setting the file to execute.
```

hold Command

Suspends the current process (if it is executing) so that you can perform debugging operations on the process.

`hold`

The `hold` command issues a `PROCESS_DEBUG_` request on the current process. After the current process is in the Hold state, Native Inspect redisplays its command prompt.

ignore Command

Enables you to specify a number of breakpoint hits to be ignored before Native Inspect reports the breakpoint has been hit.

`ignore` *`breakpoint-number ignore-count`*

Where:

`breakpoint-number`

The number of a breakpoint or catch event that you want reported.

`ignore-count`

The number of breakpoint hits that Native Inspect is to ignore before reporting the breakpoint.

ih Command

Displays information (info handler) about all signal handlers or about the handler for a specified signal. This command applies only to TNS/E native processes, not snapshot files.

Related Command: `mh` (see mh Command (page 100)).

`ih [`*`signal_name`*`]`

Where:

`signal-name`

The name of the signal about which you want information. If you do not specify a signal name, Native Inspect displays information about all the signals. Table 12 (page 101) contains a list of signal names.

The command output includes information such as the address of the signal handler, whether the signal handler is priv or not, and various flags.

Example

To display information about all the signal handlers:

```
(eInspect 3,1179): ih
Signal                  Priv/Non        Handler Mask    Flags
          SIGHUP               N        0xfffc0000      0x0     0x0
          SIGINT               N        0xfffc0000      0x0     0x0
         SIGQUIT               N        0xfffc0000      0x0     0x0
          SIGILL               N        0xfffc0000      0x0     0x0
          SIGURG               N        0xfffc0001      0x0     0x0
         SIGABRT               N        0xfffc0000      0x0     0x0
           SIGIO               N        0xfffc0001      0x0     0x0
          SIGFPE               N        0xfffc0000      0x0     0x0
         SIGKILL               N        0xfffc0000      0x0     0x0
         SIGSEGV               N        0xfffc0000      0x0     0x0
        SIGWINCH               N        0xfffc0001      0x0     0x0
         SIGPIPE               N        0xfffc0000      0x0     0x0
         SIGALRM               N        0xfffc0000      0x0     0x0
         SIGTERM               N        0xfffc0000      0x0     0x0
         SIGUSR1               N        0xfffc0000      0x0     0x0
         SIGUSR2               N        0xfffc0000      0x0     0x0
         SIGCHLD               N        0xfffc0001      0x0     0x0
         SIGRECV               N        0xfffc0001      0x0     0x0
         SIGSTOP               N        0xfffc0000      0x0     0x0
         SIGTSTP               N        0xfffc0000      0x0     0x0
       SIGMEMERR               N        0xfffc0000      0x0     0x0
        SIGNOMEM               N        0xfffc0000      0x0     0x0
       SIGMEMMGR               N        0xfffc0000      0x0     0x0
          SIGSTK               N        0xfffc0000      0x0     0x0
      SIGTIMEOUT               N        0xfffc0000      0x0     0x0
        SIGLIMIT               N        0xfffc0000      0x0     0x0
         SIGCONT               N        0xfffc0001      0x0     0x0
         SIGTTIN               N        0xfffc0000      0x0     0x0
         SIGTTOU               N        0xfffc0000      0x0     0x0
         SIGABND               N        0xfffc0000      0x0     0x0
```

in Command

Displays instructions at the specified location. The `I` command is Debug-compatible.

`i {{`*`native-address`*` [`*`count`*`]} | `*`function-name`*`}`

Where:

`native-address`

The address at which you want to display instructions. See Syntax of native-address (page 61).

`count`

The number of instruction bundles to display (the three instructions that can be executed in a single CPU cycle. The default value is 1.

`function-name`

The name of the function in the source whose instructions you want to display.

Alternate methods of displaying instructions are:

- Using the `x` command in the form: `x/i $pc`. (See x Command (page 125).)
- Using the `display` command in the form: `display/i $pc`. (See display Command (page 76).)

Example

To display instructions:

```
(eInspect 1,700): i 0x70001c80 5
eInspect 1,700):i 0x70001c80 5
Dump of assembler code from 0x70001c80:0 to 0x70001cd0:0:
;;; File: \PELICAN.$DATA3.SUBVOL.GARTESTC
   118                {
```

```
0x70001c80:0 <call9>:   [MMI]       alloc r34=ar.pfs,19,16,0
0x70001c80:1 <call9+0x6>:                   adds r12=-128,r12
0x70001c80:2 <call9+0xc>:                   mov r36=r1;;
0x70001c90:0 <call9+0x10>:      [MII]       nop.m 0x0
0x70001c90:1 <call9+0x16>:                  mov r35=b0
0x70001c90:2 <call9+0x1c>:                  mov r37=r32
0x70001ca0:0 <call9+0x20>:      [MMB]       mov r38=r33
0x70001ca0:1 <call9+0x26>:                  adds r39=80,r12
0x70001ca0:2 <call9+0x2c>:                  nop.b 0x0;;
0x70001cb0:0 <call9+0x30>:      [MII]       st8 [r39]=r37
0x70001cb0:1 <call9+0x36>:                  adds r40=88,r12
0x70001cb0:2 <call9+0x3c>:                  nop.i 0x0;;
0x70001cc0:0 <call9+0x40>:      [MII]       st8 [r40]=r38
   120                  printf("%s q = %d\n",string,q);
0x70001cc0:1 call9+0x46>:                  addl r41=96,r1
0x70001cc0:2 call9+0x4c>:                  nop.i 0x0;;
End of assembler dump.
```

info Command

Displays information about the target being debugged. No aliases are accepted for the `info` command.

`info` *`attribute`*

Where:

`attribute`

The value of *`attribute`* is one of the following:

`address` *`symbolic-name`*
: Describes where the specified symbol is stored.

`all-registers`
: Lists all registers and their contents for the currently selected frame. Listing includes floating-point registers.

`architecture`
: Lists information about the target architecture.

`args`
: Lists argument variables of the currently selected stack frame.

`breakpoints`
: Lists information about all user-defined breakpoints, including per-process breakpoints, global breakpoints, catch events and MABs.

`copying`
: Lists the conditions for redistributing copies of Native Inspect.

`display`
: Lists the expressions on the automatic display list, which are displayed when the program being debugged stops.

`dll`
: Prints information about each loadfile (program file and DLL) associated with the current process. The listing includes the loadfile name, its preferred load address, and its actual address.

`files`
: Prints names of targets and files being debugged.

`frame` *`frame-number`*

Prints information about the current or the specified stack frame. To display frame numbers, use the `bt` command. The frame at which execution is currently halted is numbered 0, and frame numbers continue consecutively to the base frame, from which execution began.

`functions` *`regular-expression`*

Prints all function names or those matching the specified regular expression.

`handle` *`signal-number`*

Displays the response that Native Inspect gives when the current program receives various signals.

`line`

Displays the starting and ending addresses of the code corresponding to the specified source line. See Syntax of locspec (page 59).

`locals`

Prints the values of local variables of the currently selected stack frame. For COBOL programs, this option prints all variables within the scope of the program unit, including variables declared with the GLOBAL attribute.

`process`

Prints information about the process being debugged, which includes displaying the object file and any DLLs in use, along with their timestamps.

`program`

Prints execution status of the program being debugged.

`registers`

Lists all integer registers and their contents.

`scope` *`locspec`*

Displays information about the local variables and argument variables for the specified scope. See Syntax of locspec (page 59).

`sessions`

Prints information about all processes being debugged by the current Native Inspect process.

`set`

Shows all settings of Native Inspect; this is equivalent to the show command.

`signals` *`signal-number`*

Displays the response that Native Inspect gives when the current program receives the specified signal.

`source`

Prints information about the current source file.

`stack`

Prints a backtrace of the stack. See also the `bt` command (bt (tn) Command (page 69)).

`symbol`

Describes what symbol is at the specified address. See the section titled Syntax of native-address (page 61).

`symbol-files`

Prints the names of all the symbol files visible to the current process, including both per-process and global symbol files.

`target`

Prints information about the target being debugged.

`types` *`regular-expression`*
: Prints all type names, or those matching the specified expression.

`variables` *`regular-expression`*
: Prints all global and static variable names, or those matching the specified expression.

`warranty`
: Prints the various types of warranty that Native Inspect users do not have.

Examples

- To display frame information:

```
(eInspect 2,676): info frame
Stack level 0, frame at 0x6ffffe50:
 ip = 0x70002300:0 in pcbDataStructs_initialize
    (\SIERRA.$YOSE1.SYMBAT1.SCXXTST:340); saved ip 0x700037d0:0
 called by frame at 0x6ffffe80
 source language c.
 Arglist at 0x6ffffe50, args:
 Locals at 0x6ffffe50, Previous frame's sp is 0x6ffffe50
 Saved registers:
  gr32 at 0x6e0000c0, gr33 at 0x6e0000c8, gr34 at 0x6e0000d0,
  gr35 at 0x6e0000d8, gr36 at 0x6e0000e0, gr37 at 0x6e0000e8,
  gr38 at 0x6e0000f0
```

- To display the values of all local variables:

```
(eInspect 3,663): info locals
pcb = (PCB_t *) 0x80048a0
pcbHandle = (PCBHandle_t *) 0x8004ed0
```

- To display information about current breakpoints:

```
(eInspect 1,325):  info break
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x70001672 in main
                            at \SIERRA.$YOSE1.SYMBAT1.SCXXTST:216
2   breakpoint     keep y   0x70003700 in test_complexTypes
                            at \SIERRA.$YOSE1.SYMBAT1.SCXXTST:420
```

```
(eInspect 1,325):  info break
Num Type           Disp Enb Glb Address    What
2   breakpoint     keep y   n   0x70003be0 at PARA1 in MAIN-PROGRAM
                                at \SIERRA.$DIVA.CBEX.CBEXAM1:75
3   breakpoint     keep y   n   0x70004440 at PARA2 OF SECTION2 in MAIN-PROGRAM
                                at \SIERRA.$DIVA.CBEX.CBEXAM1:97
4   breakpoint     keep y   n   0x70004a20 at PARA2 OF SECTION3 in MAIN-PROGRAM
                                at \SIERRA.$DIVA.CBEX.CBEXAM1:109
```

The following table defines the column headers in the `info break` command:

`Num`	The number of the breakpoint, starting with 1
`Type`	The type of breakpoint
`Disp`	Disposition, either keep, del(ete), or dis(able)
`Enb`	Enabled, either y or n (yes or no)
`Glb`	Global, either y or n (yes or no)
`Address`	The address of the breakpoint
`What`	The file or module that contains the breakpoint. For COBOL programs, Paragraph/Section information is also displayed if the breakpoint is placed on the paragraph or section.

- To display the address range of a line:

```
(eInspect 3,663): info line
Line 341 of "\SIERRA.$YOSE1.SYMBAT1.SCXXTST"
   starts at address 0x70002320:0 <pcbDataStructs_initialize>
   and ends at 0x700023c0:0 <pcbDataStructs_initialize+160>.
```

- To display registers for a frame:

```
(eInspect 3,663): info frame 1
Stack frame at 0x6ffffe80:
 ip = 0x70003870:0 in test_complexTypes (\SIERRA.$YOSE1.SYMBAT1.SCXXTST:425);
    saved ip 0x700016b0:0
 called by frame at 0x6ffffef0, caller of frame at 0x6ffffe50
 source language c.
 Arglist at 0x6ffffe80, args:
 Locals at 0x6ffffe80, Previous frame's sp is 0x6ffffe80
 Saved registers:
  gr32 at 0x6e000068, gr33 at 0x6e000070, gr34 at 0x6e000078,
  gr35 at 0x6e000080, gr36 at 0x6e000088, gr37 at 0x6e000090,
  gr38 at 0x6e000098, gr39 at 0x6e0000a0, gr40 at 0x6e0000a8,
  gr41 at 0x6e0000b0, gr42 at 0x6e0000b8
```

- To list information about currently loaded DLLs, including the program loadfile:

```
(eInspect 3,331): info dll
Loadfiles:
\PELICAN.$DATA3.COBBAT.XCS000D0: (symbol file:\PELICAN.$DATA3.COBBAT.XCS000D0) (
PIC ELF PROG)
  Text          : 0x0000000070000000 (size: 0x25000)
  Data          : 0x0000000008000000
  UID           : 21576679|146
\PELICAN.$SYSTEM.SYS00.ZCOBDLL:(PIC ELF DLL, Public LIB)
  Text          : 0xfffffffffeb10000 (size: 0xc8000)
  Data          : 0x000000006db50000
  UID           : 12085384|6
\PELICAN.$SYSTEM.SYS00.ZCREDLL:(PIC ELF DLL, Public LIB)
  Text          : 0xffffffffff630000 (size: 0x7b000)
  Data          : 0x000000006dd60000
  UID           : 12085951|6
\PELICAN.$SYSTEM.SYS00.MCPDLL:(PIC ELF DLL, Implicit LIB, May NOT set BPTs)
  Text          : 0xffffffffe1000000 (size: 0x118000)
  Constant Data : 0xffffffffe1400000
  Gateway       : 0xffffffffe1500000
  UID           : 12043755|6
\PELICAN.$SYSTEM.SYS00.INITDLL:(PIC ELF DLL, Implicit LIB, May NOT set BPTs)
  Text          : 0xffffffffe4000000 (size: 0x167c000)
  Constant Data : 0xffffffffe8000000
  Gateway       : 0xffffffffe8100000
---Type <return> to continue, or q <return> to quit---
  UID           : 12031597|6
```

- To display the contents of the registers (output is several screens long):

```
(eInspect 2,647): info reg
  pr0: 0x1
  pr1: 0x0
  pr2: 0x0
  pr3: 0x0
  pr4: 0x0
  pr5: 0x0
  pr6: 0x0
  pr7: 0x0
  pr8: 0x1
  pr9: 0x0
 pr10: 0x0
 pr11: 0x0
 pr12: 0x1
 pr13: 0x0
 pr14: 0x1
...
 gr31: 0x400000000000038c
  br0: 0x0
```

```
br1: 0x0
br2: 0x0
br3: 0x0
br4: 0x0
br5: 0x0
br6: 0xfffffffe18a3840
br7: 0xfffffffe1555440
 ip:       0xfffffffe2207040
cfm: 0x0
 ra: 0x0
 sp: 0x6fffff50
psp: 0x6fffff50
bsp: 0x6e000000
 lc: 0x0
 ec: 0x0
```

info Command (memory leak detection)

Displays commands to debug memory problems.

`info` *`attribute value`*

Where:

`attribute value`

Where *`attribute value`* is one of the following:

`corruption`

Checks for corruption in the currently allocated heap blocks. In addition, it lists the potential in-block corruptions in all the freed blocks.

`heap`

Displays a heap report, listing information such as the start of heap, end of heap, heap size, heap allocations, size of blocks, and number of instances. The report shows heap usage at the point you use the `info heap` command. The report does not show allocations that have already been freed. For example, if you make several allocations, free them all, and then use `info heap`, the result does not show any allocations.

`heap` *`filename`*

Writes heap report output to the specified file.

`heap` *`idnumber`*

Produces detailed information on the specified heap allocation including the allocation call stack.

`heap-interval` *`filename`*

Creates the report of heap growth. The data for each interval has the start and end time of the interval. If *`filename`* is mentioned, a detailed report is written in the file.

`leaks`

Displays a leak report, listing information such as the leaks, size of blocks, and number of instances.

`leaks` *`filename`*

Writes the complete leak report output to the specified file.

`leaks` *`leaknumber`*

Produces detailed information on the specified leak including the allocation call stack.

NOTE: The memory debugging feature is not available when the application is stopped in a dynamic-link library (DLL).

Examples

- To obtain a heap profile, perform the following steps:

  1. Run the debugger, load the program, and issue the `set heap-check on` command:

     For 32-bit application:

     ```
     TACL> rund minheap /lib $system.sys00.zrtcdll/ executable arguments
     (eInspect 0,248)set heap-check on
     ```

     For 64-bit application:

     ```
     TACL> run –minheap -lib=/G/system/sys00/yrtcdll executable arguments
     (eInspect 0,248)set heap-check on
     ```

     See set heap-check Command (memory leak detection) (page 115) for a description of the `set heap-check on` command.

  2. Set a breakpoint by entering the following command:

     ```
     (eInspect 0,248)b probepoint
     ```

     where *`probepoint`* is some interesting point in the application being debugged.

  3. Run the program by entering the following command:

     ```
     (eInspect 0,248)run
     ```

  4. When the program is stopped at a breakpoint, enter the following `info heap` command:

     ```
     (eInspect 0,248)info heap
     ```

     The following output is displayed:

     ```
     Analyzing heap ...done

     Actual Heap Usage:
     Heap Start =0x40408000
     Heap End =0x4041a900
     Heap Size =76288 bytes

     Outstanding Allocations:
     41558 bytes allocated in 28 blocks

     No. Total bytes Blocks Address      Function
     0   34567         1    0x40411000    myfunc()
     1    4096         1    0x7bd63000       bar()
     2    1234         1    0x40419710       baz()
     3     245         8    0x404108b0       boo()
     [...]
     ```

5. To view a specific allocation, specify the allocation number as an argument to the `info heap` command. For example:

```
(eInspect 0,248)info heap 1
4096 bytes at 0x7bd63000 (9.86%of all bytes allocated)
in bar ()at test.c:108
in main ()at test.c:17
in _start ()
in $START$()
```

   When multiple blocks are allocated from the same call stack, Native Inspect displays additional information similar to the following:

```
(eInspect 0,248)info heap 3
245 bytes in 8 blocks (0.59% of all bytes allocated)
These range in size from 26 to 36 bytes and are allocated
in boo ()
in link_the_list ()at test.c:55
in main ()at test.c:13
in _start ()
```

   You can control the stack frames that are collected for reporting at any allocation point. For more information on this feature, see the *Debugging Dynamic Memory Usage Errors Using HP WDB* white paper located at the HP WDB Documentation webpage: http://www.hp.com/go/WDB.

- To use the `info heap` command with the `min-heap-size` filter setting.

NOTE: `min-heap-size` is described under the set heap-check Command (memory leak detection) (page 115).

Sample program:

```
#include stdio.h
#include stdlib.h
main()
{
int i, *arr[1000 ];
for (i=0; i < 1000; i++)
arr[i] = malloc (49);
malloc (30);
set_brkpt_here(0)
exit(0);
}
```

Sample debugging session:

For 32-bit application:

```
TACL> rund minheap /lib $system.sys00.zrtcdll/
(eInspect 0,248)b set_brkpt_here
(eInspect 0,248)set heap-check min-heap-size 31
(eInspect 0,248)run
(eInspect 0,248)info heap
Analyzing heap …
49000 bytes allocated in 1000 blocks
No.    Total bytes     Blocks     Address     Function
0         49000          1000    0x4044eff0    main()
```

For 64-bit application:

```
TACL> run –minheap –lib=/G/system/sys00/yrtcdll/
(eInspect 0,248)b set_brkpt_here
(eInspect 0,248)set heap-check min-heap-size 31
(eInspect 0,248)run
(eInspect 0,248)info heap
Analyzing heap …
49000 bytes allocated in 1000 blocks
```

```
No.    Total bytes     Blocks     Address     Function
0         49000          1000    0x4044eff0    main()
```

- To view the leak profile, perform the following steps:
  1. Run the debugger, load the program, and issue the `set heap-check leaks on` command:

     For 32-bit application:

     ```
     TACL> rund minheap /lib $system.sys00.zrtcdll/  executable arguments
     (eInspect 0,248)set heap-check leaks on
     ```

     For 64-bit application:

     ```
     TACL> run -minheap -lib=/G/system/sys00/yrtcdll executable arguments
     (eInspect 0,248)set heap-check leaks on
     ```

     See set heap-check Command (memory leak detection) (page 115) for a description of the `set heap-check leaks on` command.
  2. Set a breakpoint by entering the following command:

     ```
     (eInspect 0,248)b probepoint
     ```

     where *`probepoint`* is some interesting point in the application being debugged.
  3. Run the program by entering the following command:

     ```
     (eInspect 0,248)run
     ```
  4. When the program is stopped at a breakpoint, enter the following `info leaks` command to display the list of memory leaks:

     ```
     (eInspect 0,248)info leaks
     ```

     The following output is displayed:

     ```
     Scanning for memory leaks...done

     2439 bytes leaked in 25 blocks

     No. Total bytes Blocks Address      Function
     0     1234          1   0x40419710   myfunc()
     1      333          1   0x40410bf8     main()
     2      245          8   0x40410838   strdup()
     [...]
     ```
  5. The debugger assigns a numeric identifier for each leak. To view a stack trace for a specific leak, specify the leak number from the list of leaks, as follows:

     ```
     (eInspect 0,248)info leak 2
     245 bytes leaked in 8 blocks (10.05% of all bytes leaked)
     These range in size from 26 to 36 bytes and are allocated in strdup ()
     in link_the_list ()at test.c:55
     in main ()at test.c:13
     in _start ()
     ```

jump Command

Continues execution of the current process at the specified line number or address.

The `jump` command changes the program counter to the specified location but does not change the current stack frame or registers.

CAUTION: Use the `jump` command with care. It can result in program failure if the target location depends on a state (such as registers) that has not been established.

```
jump locspec
```

Where:

`locspec`

The location where you want execution to stop. See Syntax of locspec (page 59).

To suspend execution at the destination location, first enter the `break` command to plant a temporary breakpoint.

Example

To jump to a specified location (line 51):

```
(eInspect 3,638): list
   44                }
   45
   46                 void call1(char *string,long long q)
   47                {
   48                  eight_byte_struct structure;
   49                  structure.a = "from call1";
   50                  structure.b = 1;
   51                  printf("%s q = %d\n",string,q);
   52                  print_and_break();
   53                  call2(structure);
(eInspect 3,638): fr 0
#0  call1 (string=0x0, q=0) at C:\cygwin\home\save\test\gartest.c:49
   49                  structure.a = "from call1";
(eInspect 3,638): b 52
Breakpoint 3 at 0x700012c0:2: file C:\cygwin\home\save\test\gartest.c, line 52.
(eInspect 6,157): c
Continuing.
From main q = 0

Breakpoint 3, call1 (string=0x70000bd0 "From main", q=0)
    at C:\cygwin\home\save\test\gartest.c:52
   52                  print_and_break();

(eInspect 3,638): jump 51
Continuing at 0x700011d0:1.
From main q = 0

Breakpoint 3, call1 (string=0x70000bd0 "From main", q=0)
    at C:\cygwin\home\save\test\gartest.c:52
   52                  print_and_break();
```

kill Command

Terminates the current process or snapshot file.

`kill`

For processes:

- If Native Inspect is debugging only one process (and if you did not explicitly start Native Inspect from the TACL command prompt), Native Inspect terminates.
- If Native Inspect is debugging multiple processes, the current process is terminated but Native Inspect remains running with no current process selected (use the `vector` command or the `attach` command to establish the current process).
- If the current process is unstoppable, the kill request is queued.

For snapshot files:

If you enter a `kill` command when examining a snapshot file, the current file is closed, but Native Inspect remains running.

list Command

Lists source code starting at the most recently listed location. Native Inspect reads source from EDIT files (file code 101) and unstructured files (file code 180).

By default, Native Inspect displays 10 lines, with the current execution location in the middle of the display, where possible. The current line is indicated by an asterisk (*) at the beginning of the line. You can use the set Command (environment) with the `listsize` option to change the list size to a value other than 10. The show Command with the `listsize` option shows the current value of the list size.

For information specific to COBOL programs, see Displaying Source Lines (page 47).

```
list [start-locspec][,end-locspec] [+|-]
```

Where:

`start-locspec`

The location at which the source code display is to begin. If you omit *`start-locspec`*, Native Inspect lists source lines relative to the current program location or the last listed source location. See Syntax of locspec (page 59).

`end-locspec`

The location at which the source code display is to end. If you omit *`end-locspec`*, Native Inspect lists 10 lines by default. See Syntax of locspec (page 59).

`+`

Directs Native Inspect to list the next *n* lines after the last line listed. The variable *n*is the value of the `listsize` option. The default is 10 lines.

`-`

Directs Native Inspect to list *n* lines preceding the last line listed. The variable *n*is the value of the `listsize` option. The default is 10 lines.

The list command warns you if it detects that the timestamp for a source file differs from the timestamp stored during compilation. For example:

```
(eInspect):
Warning: Timestamp mismatch for \SIERRA.$OS.SHENDEV.AC
Source modification time at present: 2007-04-04 11:05:09
Source modification time at compilation: 2007-04-04 11:04:07
```

Locating Source Files

If the `list` command cannot locate the source file you want, an error message is displayed that contains the compile-time location recorded in the object file:

- If the base file name is unchanged (the name of the current file is the same as the compile-time name), use the dir Command to specify the subvolume that contains the file.
- Otherwise, use the map-source-name (map) Command to map the compile-time file name to the current name. You can copy and paste the path name displayed in the error message as the left-hand argument to the map command.

For examples of locating source files, see Optionally Determining the Compilation-Time Source File Name (page 36) and Optionally Configuring a Search Path for Your Source Files (page 36).

For additional details regarding COBOL programs, see Chapter 3: Using Native Inspect With COBOL Programs.

Repeating the list Command

After you enter a list command, you can repeat it by pressing the **Enter** key at the next Native Inspect prompt. The effect is the same as if you had entered another `list` command with no parameters: listing continues with the line following the most recently listed line. This ability to repeat continues until you enter another Native Inspect command.

Examples

- Use the following command to list a source:

```
(eInspect 1,463): list
  377                   */
  378                    printf( "%s building PCBReadyList: 0, 2, 4\n", getStepPrefix( 2 ) );
  379                   PCBList_add( &PCBReadyList, PCBList.entry[0] );
  380                   pcb = PCBList.entry[0]->ref.pcb;
  381                 pcb->state = PCBState_ready;
  382                 pcb->flags.item.isReady = 1;
  383                 pcb->flags.item.isHappy = 1;
  384
  385                 PCBList_add( &PCBReadyList, PCBList.entry[2] );
  386                 pcb = PCBList.entry[2]->ref.pcb;
(eInspect 1,463): list
  387                 pcb->state = PCBState_ready;
  388                 pcb->flags.item.isReady = 1;
  389                 pcb->flags.item.isHappy = 1;
  390
  391                 PCBList_add( &PCBReadyList, PCBList.entry[4] );
  392                 pcb = PCBList.entry[4]->ref.pcb;
  393                 pcb->state = PCBState_ready;
  394                 pcb->flags.item.isReady = 1;
  395                 pcb->flags.item.isHappy = 1;
  396
```

- Use the following command to list source at a line:

```
(eInspect 1,463): list 200
  195            static PCBAttribute_t *PCBAttribute_createNonstop();
  196            static PCBAttribute_t *PCBAttribute_createSystem();
  197            static PCBHandle_t *PCBHandle_create( PCB_t *pcb );
  198            static PCBHandle_t *PCBHandle_addRef( PCBHandle_t );
  199            static void PCBList_add( PCBLink_t **list, PCBHandle_t );
  200
  201             static void pcbDataStructs_initialize();
  202
  203             static void test_complexTypes();
  204
```

- Use the following command to list source at a function:

```
eInspect 1,463): list pcbDataStructs_initialize
  335
  336                /* *************************
  337                 * pcbDataStructs_initialize
  338                 */
  339                static void pcbDataStructs_initialize()
  340                {
  341                  PCB_t          *pcb;
  342                  PCBHandle_t    *pcbHandle;
  343
  344                  printf( "%s pcbDataStructs_initialize\n", getStepPrefix(
1 ) );
```

- Use the following command to list source at an address:

```
(eInspect 3,663): list *0x700001aa0
0x70001aa0:0 is in PCBAttribute_create (\SIERRA.$YOSE1.SYMBAT1.SCXXTST:252).
  247            */
  248           static PCBAttribute_t *PCBAttribute_create()
  249           {
  250            PCBAttribute_t *pcbAttribute = (PCBAttribute_t *) malloc(
 sizeof( PCBAttribute_t ) );
  251            memset( pcbAttribute, 0, sizeof( pcbAttribute ) );
  252            return pcbAttribute;
  253
  254
  255
  256            /*
```

- COBOL example:

```
(eInspect 7,411): list MAIN
  19          ?MAIN main
  20          IDENTIFICATION DIVISION.
  21          PROGRAM-ID. main.
  22          ENVIRONMENT DIVISION.
  23          CONFIGURATION SECTION.
  24          SOURCE-COMPUTER. ABD.
  25          OBJECT-COMPUTER. ABD.
```

log Command

Turns logging on or off:

- If you specify a pathname, logging is turned on.
- If you omit the pathname, Native Inspect turns off logging and closes the current log file.

```
log [pathname|-d]
```

Where:

pathname
: The OSS pathname or Guardian file name of the log file.

  Native Inspect creates a text file (file code 101) in the current working directory or in the location you specify in the log command. If the log file already exists, Native Inspect opens it and appends output to it.

-d
: Displays the name of the log file that is currently open.

When logging is on, Native Inspect records all commands and their results in the log file.

ls (files) Command

The ls comand is an alias for the `files` command. See files (ls) Command (page 80).

mab Command

Sets a memory access breakpoint (MAB) for the current process. For each process being debugged you can set one MAB. The MAB is not assigned a breakpoint number.

When you define a MAB, you can use a low-level conditional expression, but not a high-level condition or commands to execute. The process is suspended each time the memory location is accessed in the specified manner (read, write, or change).

The `mab` command assigns an ordinal to each MAB that you set. This ordinal is displayed at the time you issue the `mab` command. You can specify this ordinal as a breakpoint identifier when using the any of the following commands to manipulate breakpoints and watchpoints:

- `enable`
- `disable`
- `delete`
- `condition`
- `commands`
- `ignore`

Entered with no arguments, the `mab` command lists information about the current memory access breakpoint.

Related Command: Use the `dmab` command (dmab Command) command to delete a memory access breakpoint.

```
mab [{*native address|variable} [size][flags] [-e locspec]]
```

Where:

**native address*
: A 32-bit or 64-bit address. See Syntax of native-address (page 61).

variable
: The name of a variable in the current process on which you want to set a memory access breakpoint. If you specify a variable, then size is optional.

size

An optional number of bytes if you specify a variable. By default, Native Inspect uses the size of the variable as the address range to watch.

If you specify `*native-address`, you must also specify `size`.

The range over which a MAB is set (that is, the combined value of *native-address* and *size*) cannot exceed a 16K-page boundary.

flags

Specifies one of the following flags:

`-c`

Specifies change access, which triggers a breakpoint when the value changes.

`-g`

Specifies a global MAB; can be set only when privileged debugging is enabled. (See Global Debugging (page 26).) You can specify both the `-g` and `-h` flags.

`-h`

Indicates a halt loop breakpoint, which can only be set by the super ID user after issuing the priv Command. Both the `-g` and `-h` flags can be set at once by the super ID user.

`-r`

Specifies read access

`-w`

Specifies write access

`-rw`

Specifies read/write access (this is the default value)

locspec

A low-level conditional expression that can be specified in any order with *flags*. See Syntax of locspec (page 59).

Examples

- To set a MAB at the current location:

```
(eInspect 1,480): mab myfunc 4
MAB 7 at 0x8000290
```

- To display the MAB:

```
(eInspect 1,480): mab
        number=7
        addr = 0x080000e0
        segid = 65535
        mabtype = 2
        global = 0
        haltloop = 0
        length = 1
There is no global MAB set
```

- To set a MAB by address:

```
(eInspect 3,880): mab *0x8000370
(eInspect 3,880): c
Continuing.
Process (3, 880) received DS_EVENT_MAB (seg:65535, addr:0x08000370, pc:0xa4bd4c1
1ea887459, len:-233208559 type:1)
0x700024e0:2 in pcbDataStructs_initialize ()
    at \SIERRA.$YOSE1.SYMBAT1.SCXXTST:350
  350 for (PCBList.count=0 ; PCBList.count < PCBLIST_MAX; PCBList.count++)
```

- To set a MAB on 4 bytes at a given address:

  ```
  (eInspect 1,480): mab *0x00000070 4
  ```

- To set a MAB on variable `j` when it changes:

  ```
  (eInspect 1,480): mab j -c
  ```

- To set a "change" MAB:

  ```
  (eInspect 1,480): mab PCBList.count -c
  ```

map-source-name (map) Command

Defines filename mapping rules between source file names at compilation time and at debug time.

Related Commands: Use the dir Command to append a directory (subvolume) to the search path used to locate source files. Use the unmap-source-name (unmap) Command to delete an existing mapping rule.

`map-source-name|map [[`*`source-name`*`] = `*`alias-name`*`] | [`*`source-prefix`*` = `*`alias-prefix`*`]`

Where:

`source-name`

The fully qualified name of the source file at compilation time. If *`source-name`* is omitted, the fully qualified name of the current source file is used. For Guardian source files, if the node name is omitted in *`source-name`*, the default node name is used. For example,

```
map $dev11.src5.hello=$test11.src5.hello
```

`alias-name`

This is either a fully qualified file name or an unqualified file name to which you are mapping the fully qualified source-name. If alias-name is unqualified, Native Inspect locates it using the subvolume search path defined by the dir Command.

`source-prefix`

A prefix (of any length) of the fully qualified name of the source file at compilation time. For Guardian source files, if the node name is omitted in *`source-prefix`*, the default node name is used.

`alias-prefix`

A prefix that will be substituted for *`source-prefix`* in source file names. For example, the following command maps `D:\nsk\T1000\src\cpu\mips\x.c` to `/h/usr/rell/src/cpu/mips/x.c`:

```
map D:\nsk\T1000=/h/usr/rell
```

All source name mappings are assigned unique mapping entry numbers. A new mapping is assigned a higher mapping entry number and takes precedence over previous mappings.

The `map` command is useful when a file name has changed in some way, whereas the `dir` command is useful when a file's directory location has changed. When you transfer files compiled on a PC or workstation to the NonStop system, the file names are often not identical.

You can use the dir Command and the `map` command in combination:

- Use the `dir` command to define the directory (subvolume).
- Use the `map` command to change the base file name.

A map-source-name with no parameters lists the existing source name mappings along with their associated mapping numbers.

Considerations

- A time-saving use of the `map = alias-name` form of the `map` command (source name omitted) is to specify this command after receiving an error from a `list` command. The current source file name is assumed, thus eliminating the need to copy-and-paste or retype the name displayed in the `list` command error message.
- New aliases created as a result of prefix matches are added to the mapping list. For example, if you enter the following command:

  `map D:\usr\T1000=/h/src`

  then a source file name of, for example, `D:\usr\T1000\src\cpu\mips\x.c` is aliased to `/h/src/src/cpu/mips/x.c.` when the source file name is listed. A subsequent `map` command displays the following aliases:

  ```
  2. D:\usr\T1000\src\cpu\mips\x.c is aliased to /h/src/src/cpu/mips/x.c (Prefix match)
  1. D:\usr\T1000 is aliased to /h/src
  ```

  If you now enter the following command:

  `mapD:\usr\T1000=/v/src`

  then the alias is added to the beginning of the alias list (with alias id 3). If you then list the source file `D:\usr\T1000\src\cpu\mips\x.c`, another new full path name alias (which is now the active alias) is added. A `map` command now displays the following:

  ```
  4. D:\usr\T1000\src\cpu\mips\x.c is aliased to /v/src/src/cpu/mips/x.c (Prefix match)
  3. D:\usr\T1000 is aliased to /v/src
  2. D:\usr\T1000\src\cpu\mips\x.c is aliased to /h/src/src/cpu/mips/x.c (Prefix match)
  1. D:\usr\T1000 is aliased to /h/src
  ```

- Higher numbered aliases take precedence over lower- numbered aliases

Examples

For more examples of mapping file names, see Optionally Determining the Compilation-Time Source File Name (page 36) and Optionally Configuring a Search Path for Your Source Files (page 36).

mh Command

Sets up signal handlers (modifies handlers) for the specified signal.

The `mh` command applies to the current process only and cannot apply to TNS emulated processes. The signal handlers can be specified as actions or as a procedure entry address.

`mh` *`signal-name`* `{SIG_IGN|SIG_ABORT|SIG_DFL|SIG_DBG|`*`native-address`*`}`

Where:

`signal-name`

The name of the signal being set up with a signal handler. See Table 12 (page 101).

`SIG_IGN|SIG_ABORT|SIG_DFL|SIG_DBG`

The signal handlers (in this case, actions), defined as follows:

- `SIG_IGN` – Ignore signal
- `SIG_ABORT` – Abort program
- `SIG_DFL` – Invoke default
- `SIG_DBG` – Invoke debugger

`native-address`

The procedure entry address at which the signal handler is set up. See Syntax of native-address (page 61).

Table 12 lists the signal names. Signal names are used in both the mh Command and the ih Command.

Table 12 TNS/E Signal Names

SIGHUP[1]	SIGSTOP[1]
SIGINT	SIGTSTP[1]
SIGQUIT[1]	SIGMEMERR
SIGILL	SIGNOMEM
SIGURG[1]	SIGMEMMGR
SIGABRT	SIGSTK
SIGIO[1]	SIGTIMEOUT
SIGFPE	SIGLIMIT
SIGKILL[1]	SIGCONT[1]
SIGSEGV	SIGTTIN[1]
SIGWINCH[1]	SIGTTOU[1]
SIGPIPE[1]	SIGABND[1]
SIGALRM[1]	SIGTERM[1]
SIGUSR1[1]	SIGUSR2[1]
ZSIGCHLD[1]	SIGRECV[1]

[1] Indicates signals that apply only in the OSS environment. Other signals apply in both the OSS and native Guardian environments.

Example

In this example of the mh command and the ih Command, the first mh command causes the signal to be reported to the debugger; the second mh command restores the default signal handler; and the ih command displays information about the signal handlers:

```
(eInspect 4,658): mh SIGNOMEM DIG_DBG
(eInspect 4,658): mh SIGCHLD SIG_DFL
(eInspect 4,658): ih
Signal                   Priv/Non        Handler Mask    Flags
          SIGHUP                N        0xfffc0000      0x0      0x0
          SIGINT                N        0xfffc0000      0x0      0x0
         SIGQUIT                N        0xfffc0000      0x0      0x0
          SIGILL                N        0xfffc0000      0x0      0x0
          SIGURG                N        0xfffc0001      0x0      0x0
         SIGABRT                N        0xfffc0000      0x0      0x0
           SIGIO                N        0xfffc0001      0x0      0x0
          SIGFPE                N        0xfffc0000      0x0      0x0
         SIGKILL                N        0xfffc0000      0x0      0x0
         SIGSEGV                N        0xfffc0000      0x0      0x0
        SIGWINCH                N        0xfffc0001      0x0      0x0
         SIGPIPE                N        0xfffc0000      0x0      0x0
         SIGALRM                N        0xfffc0000      0x0      0x0
         SIGTERM                N        0xfffc0000      0x0      0x0
         SIGUSR1                N        0xfffc0000      0x0      0x0
         SIGUSR2                N        0xfffc0000      0x0      0x0
         SIGCHLD                N        0xfffc0000      0x0      0x0
         SIGRECV                N        0xfffc0001      0x0      0x0
         SIGSTOP                N        0xfffc0000      0x0      0x0
         SIGTSTP                N        0xfffc0000      0x0      0x0
       SIGMEMERR                N        0xfffc0000      0x0      0x0
        SIGNOMEM                N        0xd          0x0      0x0
       SIGMEMMGR                N        0xfffc0000      0x0      0x0
```

```
        SIGSTK                N        0xfffc0000      0x0     0x0
    SIGTIMEOUT                N        0xfffc0000      0x0     0x0
      SIGLIMIT                N        0xfffc0000      0x0     0x0
       SIGCONT                N        0xfffc0001      0x0     0x0
       SIGTTIN                N        0xfffc0000      0x0     0x0
       SIGTTOU                N        0xfffc0000      0x0     0x0
       SIGABND                N        0xfffc0000      0x0     0x0
```

modify (mn) Command

A Debug-compatible Tcl command that changes the content of memory at *`native-address`* to *`value`*.

`modify` *`native-address`* `value {8|16|32|64}`

Where:

`native-address`

The address in memory whose contents you want to change. See Syntax of native-address (page 61).

`value`

The value you want to assign to *`native-address`*. The options `8`, `16`, `32`, and `64` specify the bit size of *`value`*.

After completion of the command, Native Inspect prints out the address that was modified, the number of bytes that were modified, and the old and the new values. If the specified *`value`* does not fit in the size specified, an error is returned.

Example

The following command returns an error because the value `0xFFFF` is larger than the specified size of 8 bits:

```
m 0x4FFFFE00 0xFFFF 8
```

next (nexti) Command

Advances program execution to the next statement or instruction, respectively. For COBOL programs, advances execution to the next verb. Execution steps over any function calls or PERFORM statements executed within the step range.

Next and nexti are similar to the step (stepi) Command.

`{next|nexti} [`*`count`*`]`

Where:

`count`

A positive integer specifying the number of statements (`next` command) or instructions (`nexti` command) to advance.

After you enter a `next` or `nexti` command, you can repeat it by pressing the **Enter** key at the next Native Inspect prompt. The effect is the same as if you had entered another `next` or `nexti` command with the same *`count`* value. This ability to repeat continues until you enter any other Native Inspect command.

Example

To step execution using the next, step, and finish commands:

```
(eInspect 1,329): next
  351                   pcb                          = PCB_create();
(eInspect 1,329): step
PCB_create () at \SIERRA.$YOSE1.SYMBAT1.SCXXTST:226
  226                 {
(eInspect 1,329): next
```

```
  227                  PCB_t *pcb = (PCB_t *) malloc( sizeof( PCB_t ) );
(eInspect 1,329): next
  228                  memset( pcb, 0, sizeof( PCB_t ) );
(eInspect 1,329): finish
Run till exit from #0  PCB_create () at \SIERRA.$YOSE1.SYMBAT1.SCXXTST:228
0x70002520:0 in pcbDataStructs_initialize ()
    at \SIERRA.$YOSE1.SYMBAT1.SCXXTST:351
  351                    pcb                            = PCB_create();
Value returned is $2 = (PCB_t *) 0x0
```

nocstm Option

The `nocstm` command-line option specifies that Native Inspect is not to execute the commands in the custom startup file named `EINSCSTM`. You can enter the `nocstm` option only when you explicitly start Native Inspect with a `RUN EINSPECT`. You cannot specify the `nocstm` option with a `RUND` command.

Refer to Reading the Custom File (page 23) for more information on the custom startup file.

```
--nocstm
```

output Command

Displays the value of the specified expression. The `output` command is identical to the `print` command except that its output is not saved in the value history buffer ($1, $2, and so on).

`output [/`*`format`*`]` *`expressions`*

Where:

`format`

An optional count, format, and size specification. See Syntax of /format (page 63).

`expression`

Evaluates the expression and displays its value. See Syntax of expression (page 62).

print Command

Evaluates and displays the value of a specified expression.

The `print` command assigns a number to each output value, and the displayed result is saved in the value history buffer. You can use the value history buffer to display the result of previous print commands; specify the number assigned by the `print` command ($1, $2, $3, and so on).

Entered with no options, redisplays the last value in the value history buffer.

Effective with the H06.14 RVU, you can invoke command line function calls in the program being debugged from the debugger command line by using the `print` command.

NOTE:

- Command line function calls cannot be used with snapshot files.
- Functions that include a breakpoint cannot be involved in a call chain initiated via the command line call mechanism.
- Command line calls are not yet available for C++, COBOL, and pTAL.
- Command line calls are not available when application is stopped in a dynamic-link library (DLL).

Special options accepted by the `print` command are:

- $ refers to the last print display.
- $$ refers to the next-to-last print display.

Related Commands: info locals, info args

```
print [format] expression
```

Where:

format
: An optional count, format, and size specification. See Syntax of /format (page 63).

expression
: Evaluates the expression and assigns its value to a variable.

  In COBOL expressions, use COBOL operators, and in C and pTAL expressions, use the standard C assignment operators, including +=, *=, and \=.

  See Syntax of expression (page 62).

Considerations

- Other functions of the `print` command (illustrated in Examples (page 105)) include the following:

Print Command Form	Function
`print (cast-type) variable`	Displays a variable formatted as a specific type (specified by a C/C++ cast expression)
`print {struct} buffer`	Displays a buffer formatted as the specified structure. Equivalent to the `DISPLAY AS` command in Inspect.
`print variable@elements`	Displays the specified number of instances of a variable. Example: `print var@2`
`print array [index] @count`	Displays a range of array elements.
`print file::variable`	Displays a static variable in a specific file or function
`print function::variable`	
`print var=exp`	Evaluates an expression and assigns the result to a variable. The expression can be any valid C expression. This command is equivalent to the following command: `set variable var = exp`
`print func (args)`	Calls a procedure, function or subroutine in the process being debugged.
`call func using arg [,arg]… ]`	

- To view an appropriately formatted SPI buffer, you must use Visual Inspect instead of Native Inspect. Native Inspect does not support the formatting of SPI buffers.
- For C/C++, character pointers are displayed until a terminating null or the configured maximum number of elements is displayed. Use the following command to control the maximum number of elements printed for strings or arrays:

  ```
  set print elements max-number
  ```

  For pTAL code, only one character of a pointer to a string is printed. An array of strings is printed as an array with each different element printed separately. If the elements are all characters, however, they are printed as one string.
- After you enter a print command, you can repeat it by pressing the **Enter** key at the next Native Inspect prompt. The effect is the same as if you had entered another print command with the same parameters. This ability to repeat continues until you enter any other Native Inspect command.

Examples

- To display constant expressions, (compared to display of eq command), use the following command:

```
(eInspect 3,638): print /x 0x6ffffe40 + (47 *7)
$9 = 0x6fffff89
(eInspect 3,638): print /c 64
$8 = 64 '@,
(eInspect 3,638): eq 64
OCT: 000100  DEC: 64     HEX: 0x0040  ASCII: '...@'
```

- To display variable addresses:

```
(eInspect 3,657): print &PCBList
$5 = (struct PCBList_s *) 0x8000320
```

- To display the address of a function:

```
(eInspect 3,663): print &PCB_addAttribute
$12 = (void (*)(PCB_t *, PCBAttribute_t *)) 0x70001820:0 <PCB_addAttribute>
```

- To display character pointers (note that character pointers are automatically dereferenced) use the following command:

```
(eInspect 3,638): print new_ptr
$11 = 0x80001c0 "In print_and_break\n"
(eInspect 3,638): set print elements 10
(eInspect 3,638): print new_ptr
$14 = 0x80001c0 "In print_a"...
```

- The following example uses the COBOL concatenation operator:

```
(eInspect 0,434): print "E" & "M"
$1 = "EM"
(eInspect 3,434): print "ABC" & "DEF"
$2 = "ABCDEF"
```

- To display pointers:

```
(eInspect 3,657): print pcb
$1 = (PCB_t *) 0x80048a0
(eInspect 3,657): print *pcb
$2 = {
  state = 0,
  flags = {
    word = 0,
    item = {
      isBad = 0,
      isReady = 0,
      isHappy = 0,
      isStarved = 0,
      waitState = 0
    }
  },
  dispatchCount = 0,
  pin = 0,
  attributeCount = 2,
  attribute = {0x8004ee0, 0x8004f40, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}
}
(eInspect 3,657): print pcb->flags.item.isHappy
$3 = 0
```

- To modify registers:

```
(eInspect 3,663):  print $gr33
$21 = 1879062640
(eInspect 3,663):  print $gr33 = 10
$22 = 10
(eInspect 3,663):  print $gr33
$23 = 10
```

```
(eInspect 3,663):  print $gr33 += 10
$24 = 20
```

- To display an instruction pointer:

```
(eInspect 3,663):  print /x $ip
$29 = 0x70002bf0
```

- Use the @ symbol to control the number of instances printed. In this example, three commands contain the @ symbol. The first command displays two instances of the entire attribute array. The second command displays three array elements starting at element 1. The third command displays 50 array elements starting at element 0:

```
(eInspect 4,782):  print pcb->attribute
 $2 = {0x8004ee0, 0x8004f40, 0x8004fa0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}

 (eInspect 4,782):  print pcb->attribute@2
 $3 = {{0x8004ee0, 0x8004f40, 0x8004fa0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, {
     0xffffffffaaaaaaaa, 0x10, 0x1, 0x80048a0, 0xffffffffaaaaaaaa, 0x40, 0x0,
     0x0, 0x0, 0x10000}}

 (eInspect 4,782):  print pcb->attribute[1]@3
 $4 = {0x8004f40, 0x8004fa0, 0x0}

 (eInspect 4,782):  print gBuffer[0]@50
 $5 = '\000' <repeats 49 times>
```

- To display a buffer formatted as a ‘C’/C++ struct or pTAL record, (use the `set print pretty` command to control display format of structures), use the following command:

```
(eInspect 3,663): print gBuffer
$7 = '\000' <repeats 15 times>, "\003\b\000N\340\b\000O@\b\000O\240",
'\000' <repeats 27 times>
(eInspect 3,663): print {PCB_t} gBuffer
$8 = {
  state = 0,
  flags = {
    word = 0,
    item = {
      isBad = 0,
      isReady = 0,
      isHappy = 0,
      isStarved = 0,
      waitState = 0
    }
  },
  dispatchCount = 0,
  pin = 0,
  attributeCount = 3,
  attribute = {0x8004ee0, 0x8004f40, 0x8004fa0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
    0x0}
}
```

- To display a range of array elements, (use the `set print array` to control display format of arrays), use the following command:

```
(eInspect 3,663): print gBuffer
$23 = "0123456789\000\000\000\000\000\003\b\000N\340\b\000O@\b\000O\240",
'\000' <repeats 27 times>
(eInspect 3,663): print gBuffer[2]@5
$24 = "23456"
```

- To display complex C/C++ data structures:

```
(eInspect 3,657):  print PCBList->entry[0]->ref.pcb->attribute->att.system
$12 = {
  privLevel = 5
}
```

- To display cast expressions:

```
(eInspect 1,463):  print pcb->flags
$4 = {
    item = {
    isBad = 0,
    isReady = 1,
    isHappy = 0,
    isStarved = 0,
    waitState = 0
  }
}(eInspect 1,463):  print /t (unsigned long) pcb->flags
$5 = 10000000000000000000000000000000
```

- To display an item from the value history list:

```
(eInspect 1,463): print *PCBList.entry[0]
$11 = {
  refCount = 2,
  ref = {
    address = 134236320,
    pcb = 0x80048a0
  }
}
(eInspect 1,463): print $11
$12 = {
  refCount = 2,
  ref = {
    address = 134236320,
    pcb = 0x80048a0
  }
}
```

- To modify variables:

```
(eInspect 3,663): print pcb->pin
$26 = 0
(eInspect 3,663): print pcb->pin=4
$27 = 4
(eInspect 3,663): print pcb->pin
$28 = 4

(eInspect 3,663): print pcb->attributeCount = PCBList.entry[1]->ref.pcb->attributeCount
$33 = 0
```

- To modify variables:

```
(eInspect 4,219): p structure3
$1 = {
  c = 0x70000c00 "From call2",
  d = 2
}
(eInspect 4,219): p structure3.d=sizeof(structure3)
$2 = 16
(eInspect 4,219): p structure3
$3 = {
  c = 0x70000c00 "From call2",
  d = 16
}(eInspect 4,219):
```

- To display the value of a COBOL expression:

```
(eInspect 7.390): print DI
$3=132
(eInspect 7,390): print DI IS NOT GREATER THAN 0
$4='T'
(eInspect 7,390): print DI GREATER THAN 0
$5='T'
(eInspect 7,390): print DI=6
$6=6
```

priv Command

Sets, unsets, or shows the privilege level of the debugging session. The privilege level in turn controls whether a user can perform privileged debugging operations such as setting breakpoints on or stepping in to privileged functions.

To use the `priv` command to turn privileged debugging on or off, you must be logged on as the super ID.

Entered with no options, displays the current priv mode status.

```
priv [on|off]
```

ptype Command

Prints the definition of a specified data type or the data type of a variable or expression.

`ptype [`*`data-type`*`|`*`variable-name`*`]`

Where:

`data-type`

The data type about which you want information. For C/C++ code, data-type can have the form `class class-name`, `struct struct-tag`, `union union-tag`, or `enum enum-tag`.

`variable-name`

The name of a variable whose data type you want displayed.

The `ptype` command is similar to the whatis Command except that `ptype` prints the detailed description, while `whatis` prints just the name of the data type.

Example

- To display the definition of the struct named `a_struct`:

```
(eInspect 7,519): ptype a_struct
type = struct type_struct {
    int i;
    char c;
}
```

- COBOL example:

```
(eInspect 7,444): ptype EM
type = PIC X(4)
(eInspect 7.444): ptype RECX
type = RECORD
       02 NATA NATIVE-4
       02 NATB NATIVE-8
(eInspect 7,444): ptype ARRX
type = NATIVE-2 OCCURS 4 TIMES
```

pwd Command

Prints the current working directory.

pwd

Example

To display the current working directory:

```
(eInspect 2,1142): pwd
Working directory \PIPPIN.$D0101.INSPECT.
```

quit (exit) Command

Ends a Native Inspect session. See also exit (quit) Command (page 79).

`{quit|exit}`

When you enter a `quit` or `exit` command, Native Inspect detaches itself from the current process and stops.

If you exit while a process is suspended, it is automatically resumed.

If breakpoints are set in a process, Native Inspect prompts you whether you want to continue. If you do, another debugger instance is started if one of the breakpoints is hit.

reg Command

A Tcl command that displays registers for the currently selected stack frame.

`reg`

Table 13 lists the names of the registers on a TNS/E native NonStop system. The info, print, and reg commands in particular use register names. You can use any C assignment operator (such as += or `-=`) when assigning a value to a register.

Table 13 TNS/E Register Names

Type of Register	Register Names
General registers	`$gr0 - $gr127`
Floating-point registers	`$fr0 - $fr127`
Predicate registers	`$pr0 - $pr63`
Branch registers	`$br0` - `$br7`
Application registers	`$bsp, $ec, $lc`
Other registers	`$ip` (instruction pointer), `$cfm` (current frame marker), `$psr` (program status register), `$sp` (stack pointer)

save Command

Creates a snapshot file (or save file) of the current TNS/E or TNS emulated process that you can debug at a later time. Snapshot files have the file code `130` and are used for offline debugging. Snapshots for TNS processes can only be read by Visual Inspect and Inspect and snapshots for 32-bit TNS/E processes can only be read by Visual Inspect and Native Inspect. Snapshots for 64-bit processes can only be read by Native Inspect.

`save` *`name`* `[`*`compression`*`] [!]`

Where:

`name`

The OSS pathname or Guardian file name , (or a file name for the default location), where the snapshot file is written.

`compression`

When debugging a 64-bit process, the `save` command allows control over the compression type applied to the file. You can specify whether or not to apply compression, and which compression algorithm to use.

It can be one of the following:

- `bzip2`
- `gzip`

  Indicates the compression algorithm type.
- `none`

  Indicates no compression is required.

NOTE: If no value is specified, the debugger may or may not compress the snapshot file based on its size.

`!`

Forces the overwriting of the specified snapshot file. If you do not include the exclamation mark(`!`), and the specified file already exists, Native Inspect reports an error.

The `save` command returns control to the target process. Consequently, when you list the `save` in a sequence of commands by using the `commands` command, the `save` command must always be the final command in the sequence.

select-frame Command

Is the "silent" version of the frame (select-frame) Command, and the syntax of the two commands is similar. Select-frame only selects a frame and does not print out information about the frame.

set Command (environment)

Sets debugging session options and environment variables. Compare with the set Command (variable), which modifies the value of a variable in the program being debugged. All the environment attributes available to the `set` command are also supported by the show Command. Some attributes, however, are supported only by the `show` command.

`set` *`attribute value`*

Where *`attribute value`* is one of the following:

`complaints` *`max-number`*

Sets the maximum number of complaints about incorrect symbols. The default value is 0 (zero).

`continue-to-main` *`{on|off}`*

Specifies whether or not execution is to automatically advance to the main program when you start a C/C++ program with `RUND` (Guardian) or `run -debug` (OSS). The default value is `on` (execution automatically advances to main()). If you specify `off`, you must set a breakpoint to stop execution at main(). For more information on setting a breakpoint at main(), see Advancing Execution to main() in C/C++ Programs (page 36).

`check` *`sub-attribute value`*

Where *`sub-attribute`* is one of the following:

- `range {on|off|warn|auto}`

  Sets range checking.

- `type {on|off|warnauto}`

  Sets type checking.

`confirm {on|off}`

Sets whether Native Inspect prompts for confirmation before performing potentially dangerous operations.

`height` *`number-lines`*

Sets the number of lines that are printed to the screen before Native Inspect issues a "Continue Output" prompt. Specifying 0 disables prompting.

`history` *`sub-attribute value`*

Sets attributes of the command history maintained by Native Inspect.

`sub-attribute` is one of the following:

- `expansion {on|off}`

Sets history expansion on command input.

- `filename {on|off}`

  Sets the name of the file in which the command history is recorded.
- `save {on|off}`

  Sets saving of the history record on exit from Native Inspect.
- `size` *`number`*

  Sets the size of the command history. The default value is 256.

`input-radix {8|10|16}`
: Sets the default input radix for entering numbers.

`language {ptal|c|c++|cobol}`
: Sets the current source language.

  When debugging pTAL using Native Inspect:

  - You must use C/C++ syntax.
  - Use the == operator for equality; use = for assignment.
  - Pointers are not automatically dereferenced; you must use the C * operator.
  - In subprocs, you must qualify variable names in the containing proc by using this syntax:

    `procname::localvarname`

    Note that PROC and SUBPROC names are listed in the stack trace in uppercase letters.
  - You can access global variables hidden by a local or sublocal by using `::`*`localvarname`*.
  - To set breakpoints on subprocs, use the dot operator to qualify the subproc name:

    `procname.subprocname`

    See Syntax of expression (page 62) for more information about using pTAL with Native Inspect.

`listsize` *`number`*
: Sets the value of the size of the list displayed by the `list` command. The default value is 10.

`max-function-matches` *`n`*
: Sets the maximum number of function matches reported by the `info functions` command. Specify an integer as the value of *n* or set the value to zero (`0`) to report all matches.

`mode {user | priv [on | off]}`
: Sets the working session as priv (privileged) mode for process debugging. Applies to individual debug sessions; restricted to the super ID (user 255,255).

`optimized-code-warning [off | on]`
: Sets a warning indicating that the debug information is incomplete. The default is set to `off`.

`optimized-loc-print` *`N`*
: Controls how many “near” locations (near to the current instruction address) are revealed to the end user when a variable is not found at the current location but has a location and value near to the current instruction.

  N is the maximum number of locations listed. The default is 3 locations. Setting the number to 0:

  ```
  set optimized-loc-print 0
  ```

  disables the feature entirely.

`output-radix {8|10|16}`
: Sets the output radix for printing of values.

`print` *`sub-attribute value`*

Where sub-attribute is one of the following:

`address {on|off}`
: sets printing of addresses.

`array {on|off}`
: Sets pretty printing of arrays (prints one field per line rather than compressing multiple fields on one line). See the Examples (page 105) following the `print` command syntax.

`dereference [{on|off}]`
: Controls the display of `char *` variables. The default is `on` , and `char *` variables are automatically dereferenced for display. When you specify the `off` option, `char *` variables are not dereferenced for display. See example of dereference usage.

`asm-demangle {on|off}`
: Sets demangling of C++ names in disassembly listings.

`cobol-arg-values {on|off}`
: For COBOL programs, sets displaying of argument values for when a breakpoint is encountered, a backtrace is done, or execution steps into a function. If this parameter is set to OFF (the default setting), Native Inspect displays only the addresses of arguments when any of these events occur.

`demangle {on|off}`
: Sets demangling of encoded C++ names when displaying symbols.

`elements` *`number`*
: Sets the limit of string chars or array elements to print. The default value is 200 characters.

`max-symbolic-offset` *`number`*
: Sets the largest offset that is printed in the `symbol+offset` form.

`null-stop {on|off}`
: Sets printing of char arrays to stop a first null char.

`object {on|off}`
: Sets printing of objects; derives type based on vtable information.

`pretty {on|off}`
: Sets pretty printing of structures (prints one field per line rather than compressing multiple fields on one line). See the following examples.

`repeats` *`number`*
: Sets threshold for repeated print elements. The default value is 10.

`sevenbit-strings {on|off}`
: Sets printing of 8-bit characters in strings as `\`*`nnn`*.

`static-members {on|off}`
: Sets printing of C++ static members.

`symbol-filename {on|off}`
: Sets printing of source file name and line numbers with `symbol.`

`symbolic-addr {on|off}`
: sets printing of the symbolic address (*`proc-name`* + *`offset`*) adjacent to the numeric address.

`union {on|off}`
: Sets printing of unions interior to structures.

`vtable {on|off}`
: Sets printing of C++ virtual function tables.

`radix {8|10|16}`
: Sets the input and output number radices.

`symbol-reloading {on|off}`
: Sets dynamic symbol table reloading multiple times in one run.

`verbose {on|off}`
: Sets verbosity (information about progress is displayed as a command executes).

`width` *`number`*
: Sets the number of characters Native Inspect expects in a line.

Examples

- To set pretty printing on and off:

```
(eInspect 4,770): set print pretty off
(eInspect 4,770): print pcb
$2 = (PCB_t *) 0x0
(eInspect 4,770): c
Continuing.
Breakpoint 3, pcbDataStructs_initialize ()
    at \SIERRA.$YOSE1.SYMBAT1.SCXXTST:362
  362  PCB_addAttribute( pcb, PCBAttribute_createSystem( 5 ) );
(eInspect 4,770):  print *pcb
$3 = {state = 0, flags = {word = 0, item = {isBad = 0, isReady = 0,
      isHappy = 0, isStarved = 0, waitState = 0}}, dispatchCount = 0, pin = 0,
  attributeCount = 0, attribute = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
    0x0, 0x0}}
(eInspect 4,770):  set print pretty on
(eInspect 4,770):  print *pcb
$4 = {
  state = 0,
  flags = {
    word = 0,

    item = {
      isBad = 0,
      isReady = 0,
      isBad = 0,
      isReady = 0,
      isHappy = 0,
      isStarved = 0,
      waitState = 0
    }
  },
  dispatchCount = 0,
  pin = 0,
  attributeCount = 0,
  attribute = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}
}
```

- To set pretty printing for arrays:

```
(eInspect 4,770):  set print array off
(eInspect 4,770):  print pcb->attribute
 $5 = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}
(eInspect 4,770):  set print array on
(eInspect 4,770):  print pcb->attribute
 $6 =   {0x0,
   0x0,
   0x0,
   0x0,
   0x0,
   0x0,
   0x0,
   0x0,
   0x0,
   0x0}
```

- To set the number of elements to print:

```
(eInspect 4,774):  set print elements 5
(eInspect 4,774):  print pcb->attribute
 $5 =   {0x0,
   0x0,
   0x0,
   0x0,
   0x0...}
```

- To set print `dereference`, use the following commands:

```
(eInspect 4,774):  list 1
     1   char * hw = "hello world!";
     2   main()
     3   {
     4     printf("%s\n", hw);
*    5     }
(eInspect): show print dereference
Suppress dereferencing of char * is on.
(eInspect): set print dereference
(eInspect):  print hw
$3 = 0x70000900 "hello world!"
(eInspect): set print dereference off
(eInspect): print hw
$4 = 0x70000900
```

- To set `max-function-matches`, use the following commands:

```
(eInspect): set max-function-matches 5
(eInspect): show max-function-matches
Maximum number of function matches reported by info functions
(set to zero to report all matches) is 5.
(eInspect): set max-function-matches 0
(eInspect): show max-function-matches
Maximum number of function matches reported by info functions
(set to zero to report all matches) is unlimited.
```

- Printing an optimized variable shows where the variable has not been optimized away and is available. This is intended mostly for "live target" (non-snapshot) debugging. You can then either move forward in the execution or place a break at an earlier point to find a line/address where the value of a variable prints successfully. The number of closest locations is set to 3, but can be controlled using `set optimized-loc-print`:

```
(eInspect 3,904):p i
Due to optimization, the address/value of "i" is unknown for the
current location.
Closest location(s) where it is available:
1) 0x70001452 to 0x70001540 (in line 84 to 88) in register $r39
2) 0x700017c1 to 0x70001890 (in line 99 to 103) in register $r32
```

set Command (variable)

Evaluates an expression and assigns the resulting value to a variable.

`set [variable]` *`var-name {expression|value}`*

Where:

`var-name`

The name of a variable in the current process.

`expression`

is an expression whose value you want to be assigned to *`var-name`*. See Syntax of expression (page 62).

value

A value you want to assign to *var-name*.

Consideration

Although the keyword `variable` is optional, you must include it if the name of the variable you are setting conflicts with the name of an option supported by the set Command (environment).

For example, a variable name *p* conflicts with the `print` option of the `set` command, and entering a `set` command without the `variable` keyword results in an error:

```
(eInspect 1,187): set p=6
Undefined set print command: "=6".  Try "help set print".
```

In this case, you must use one of the following alternatives:

- Modify the value of the variable by using the assignment operator with the `print` command:

  ```
  print p=6
  ```

- Include the `variable` option in your `set` command:

  ```
  set variable p=6
  ```

set heap-check Command (memory leak detection)

Displays settings for the commands that debug memory problems.

`set heap-check` *`attribute value`*

Where:

`attribute value`

Where *`attribute value`* is one of the following:

`[on | off]`

Toggles the capability for detection of leaks, heap profiles, bounds checking, and checking for double free.

`block-size` *`num-bytes`*

Instructs Native Inspect to stop the program whenever it tries to allocate a block larger than *`num-bytes`* in size.

`frame-count` *`num`*

Controls the depth of the call stack collected. Larger values increase run time. The default value is four (4) stack frames.

`free [on | off]`

When set to `on`, forces Native Inspect to stop the program when it detects a call to `free()` with an improper argument, a call to `realloc()` that does not point to a valid currently allocated heap block, or a call to free a block of memory that has been corrupted.

`heap-size` *`num-size`*

Instructs Native Inspect to stop the program whenever it tries to increase the program heap by at least *`num-bytes`*.

`leaks [on | off]`

Controls Native Inspect memory leak checking.

`min-heap-size` *`num`*

This option reports the heap allocations that exceed the specified number (*`num`*) of bytes based on the cumulative number of bytes that are allocated at each call-site, which is inclusive of multiple calls to `malloc` at a particular call site.

`min-leak-size` *`num`*

Collects a stack trace only when the size of the leak exceeds the number of bytes you specify for this value. Larger values improve run-time performance. The default value is zero (0) bytes.

`string [on | off]`

Toggles validation of calls to `strcpy`, `strncpy`, `memcpy`, `memccpy`, `memset`, `memmove`, `bzero`, and `bcopy`. Native Inspect validates calls to `strcat` and `strncat`.

Examples

- To enable heap checking, enter the following:

  For 32-bit application:

  ```
  TACL> rund minheap /lib $system.sys00.zrtcdll/  executable arguments
  (eInspect 0,248)set heap-check on
  ```

  For 64-bit application:

  ```
  TACL> run –minheap –lib=/G/system/sys00/yrtcdll  executable arguments
  (eInspect 0,248)set heap-check on
  ```

  To obtain a snapshot heap profile, run the debugger and load the program, follow the steps as shown in the examples under the info Command (memory leak detection) (page 90).

- To enable leak checking, enter the following:

  For 32-bit application:

  ```
  TACL> rund minheap /lib $system.sys00.zrtcdll/ executable arguments
  (eInspect 0,248)set heap-check leaks on
  ```

  For 64-bit application:

  ```
  TACL> run –minheap –lib=/G/system/sys00/yrtcdll executable arguments
  (eInspect 0,248)set heap-check leaks on
  ```

  To view the leak profile, run the debugger and load the program, follow the steps as shown in the examples under the info Command (memory leak detection) (page 90).

show Command

Displays either the environment settings for Native Inspect or the value of a variable in the program being debugged.

```
show [attribute|{history|print|check} [sub-attr]]
```

Where:

`attribute`

This is typically one of the same attributes as for the set Command (environment).

The `history`, `print`, and `check` attributes are as described for the set Command (environment).

Attributes supported only by the `show` command (not by the `set` command) include the following:

`commands`

Displays the history of commands. Displays ten lines of commands.

`copying`

Displays conditions for redistributing copies of Native Inspect.

`directories`

Displays current search path for finding source files. Native Inspect always searches the compilation directory (`$cdir`) and the current working directory (`$cwd`). You can set additional subvolumes for searching by using the dir Command.

`heap-check`

Displays all current settings for memory checking.

`listsize`

Displays the current value of the size of the list displayed by the `list` command.

`user` *`commandname`*

Displays definitions (but not documentation) of user-defined commands. If *`commandname`* is not specified, this displays the definitions for all user-defined commands.

When user-defined commands are executed, the commands of the definition are not printed. An error in any command stops execution of the user-defined command.

If used interactively, commands that would ask for confirmation proceed without asking when used inside a user-defined command. Many commands that normally print messages to say what they are doing omit the messages when used in a user-defined command.

`version`

Displays the version of Native Inspect, of its direct parent GDB, and of the Tcl scripting language.

`warranty`

Displays the types of warranty for Native Inspect.

Example

To display the source search path:

```
(eInspect 1,-2): show directories
Source directories searched: $d0117.tests:$cdir:$cwd
```

snapshot Command

Opens a TNS/E native process snapshot file (file code 130) for analysis.

`snapshot` *`pathname`*

`pathname`

The OSS pathname or Guardian file name of a TNS/E native snapshot file (with file code 130) that you want to open.

You cannot enter a `snapshot` command if Native Inspect already has a debugging target open (a native process or snapshot file). You must close the existing target before using the `snapshot` command.

source Command

Reads commands from the specified input file. The `source` command is the WDB and GDB equivalent of the `OBEY` command supported by Inspect and in many command interpreters on HP NonStop systems.

`source` *`pathname`*

Where:

`pathname`

The OSS pathname or Guardian file name of a file containing Native Inspect commands.

Example

The following example reads commands from the file `$disk.mysubvol.ninspcmd` (assuming your current default directory is `$disk.mysubvol`). However, the initial attempt specifies the wrong file name, invoking an error message:

```
(eInspect) source inspcmd
Can not locate command file ninspcmd(eInspect)
source ninspcmd
```

Suppose you have the following commands in a file:

```
list
show language
info target
show dir
```

If you run a program and source this file in, you will see the following:

```
(eInspect 1,749): source $data3.subvol.ex
    44                void call9(char *string,long long q);
    45
    46               void print_and_break (void) {
    47
    48                 char *new_ptr = "In print_and_break\n";
    49                 int z = 7;
    50                 printf ("About to call DEBUG'\n");
    51               }
    52               void  main (void) {
*   53                 char *local_ptr = "From main";
The current source language is "auto; currently c".
Symbols from "\PELICAN.$DATA3.SUBVOL.GARTEST".
NSK child process:
        Using the running image of child process 749.
Source directories searched: $cdir:$cwd
```

step (stepi) Command

Advances program execution by one source statement or by a specified number of statements. For COBOL programs, execution advances to the next verb. Execution steps "in" any function calls or PERFORM statements that are executed within the step range.

If a function call is made within the stepping range, the call is followed and execution suspended after the function's prolog code is executed. Execution transparently steps though any run-time environment functions, such as import stubs, for which stack unwind information is not present.

The `stepi` command advances program execution similarly, but the units of stepping is instructions. Also, function prolog code is not automatically executed by the `stepi` command. You will have to step through the prolog code before the stack frame is properly initialized.

The `step` and `stepi` commands are similar to the next (nexti) Command.

`{step|stepi} [`*`count`*`]`

Where:

`count`

A positive integer, the number of statements (`step` command), or instructions (`stepi` command) that you want to advance.

For both `step` and `stepi`, if the program calls a privileged function but the set mode command has not been used to enable privileged debugging, execution steps over the function.

After you enter a `step` or `stepi` command, you can repeat it by pressing the **Enter** key at the next Native Inspect prompt. The effect is the same as if you had entered another `step` or `stepi` command with the same count value. This ability to repeat continues until you enter any other Native Inspect command.

Example

See the example of stepping execution for the next (nexti) Command (page 102).

switch Command

Transfers the current process to either Visual Inspect or Inspect, as appropriate:

- Transfers a TNS/E native process to Visual Inspect.

  You must be running a Visual Inspect client (on Windows) connected to the NonStop system using the same user ID as the TNS/E native process.
- Transfers a TNS process to Inspect.

```
switch
```

After you enter a `switch` command, Native Inspect suspends command prompting until one of the following happens:

- The process is transferred back to Native Inspect.
- You press the **Break** key.
- The process terminates.

Native Inspect continues to maintain the associated state for the process until the process terminates or is transferred back to Native Inspect.

Breakpoint attributes are not passed between debuggers. For more information, see Switching Between Debuggers (Inspect and Visual Inspect) (page 31).

symbol-file (symbol) Command

Opens a TNS/E native code file (with file code 800) for building internal symbol tables. The new symbol table data is added to the existing data.

- Entered with no symbol-file name, prompts you before deleting all symbol files with per-process scope associated with the current process.
- Entered with only the `-g` option, prompts you before deleting all global symbol files associated with the current process.

symbol is an alias for symbol-file.

```
{symbol|symbol-file} [-g] [-readnow] pathname
```

Where:

`-g`

Loads a symbol file that has global scope. Symbols are visible to all processes being debugged.

Entered without the `-g` option, loads a symbol file that has per-process scope. Symbols are visible only to the current process.

If there is no current process, the added symbol file has global scope by default, and the `-g` option is optional.

You can add the same file with per-process scope and global scope.

`-readnow`

Expands the symbol table immediately rather than incrementally as needed.

`pathname`

The OSS pathname or Guardian file name of the TNS/E native code file that Native Inspect is to open.

The `symbol` command reads in the symbols for the specified loadfile at the corresponding loadfile’s actual load address, if it can be determined. Otherwise, the symbols are read in at the loadfile’s preferred load address, as determined at static link time, and recorded in the loadfile’s header.

Related Commands

- To delete symbols data for a specific file, use the unload-symbol-file Command.
- To load a symbol file at a specific address, use the add-symbol-file Command.
- To list the symbol files currently loaded, use the info Command with the `symbol-files` option.

For more information, see Optionally Loading Symbols Information (page 34).

Example

To load a symbol file and display information about the symbol file:

```
(eInspect 4,798): symbol-file $system.sys00.nNonStopsym
Reading symbols from $system.sys00.nnsksym...done.
(eInspect 4,798): info symbol-files
Loaded Symbol Files:
$system.sys00.nnsksym (0x50000000) (user loaded)
\PIPPIN.$D0117.SYMBAT1.XC89TST0 (0x70000000)
```

tbreak Command

See the description of the break (tbreak) Command (page 67).

tj Command

Traces the stack relative to the location stored in a jump buffer. Initialize jmp_buf by calling setjmp (in C/C++) to define a location to which a subsequent call to longjmp can branch. The `tj` command lists frames from the point at which the program calls setjmp to initialize the buffer until the base of the stack is reached.

`tj` ***`native-address`***

Where:

`native-address`

The address that contains the jump buffer.

NOTE: Unexpected results can occur if the native address specified does not correspond to a valid native jump buffer. See Syntax of native-address (page 61).

tu Command

Traces the stack relative to the location stored in a `ucontext` buffer. Set up `ucontext_t*` as the third parameter of a signal handler function (in C/C++ or pTAL). The `tu` command lists stack traces from a `ucontext` buffer contained at the specified address.

`tu` ***`native-address`***

Where:

`native-address`

The address that contains the `ucontext` buffer.

NOTE: Unexpected results can occur if the native address specified does not correspond to a valid `ucontext`. See Syntax of native-address (page 61).

Examples

- To obtain the name of the jump buffer (in this case, `env`), and then trace the stack relative to the jump buffer:

```
(eInspect 1,1028): info var
All defined variables:
```

```
File \SIERRA.$OS.VIVTEST.STEST12:
char NULL[16];
char SEEK_CUR[19];
char SEEK_END[19];
char SEEK_SET[19];
---Type <return> to continue, or q <return> to quit---
char TDMSIGH_TNS_BE_FILLER[39];
char TDMSIGH_TNS_E_FILLER[51];
char _GUARDIAN_HOST[25];
char _GUARDIAN_TARGET[27];
char _TANDEM_ARCH_[24];
char _TANDEM_SOURCE[25];
char __INT32[18];
char __TANDEM[19];

char __XMEM[17];
char __size_t_DEFINED[26];
jmp_buf env;
int errno;
char int64_t[26];

Non-debugging symbols:
        08000230  _initz
        700008d0  _CTORS
        700008d0  _ctors
        700008d8  _DTORS
        700008d8  _dtors
        700008e0  _termz
(eInspect 1,1028):
(eInspect 1,1028): p &env
$1 = (jmp_buf *) 0x8000350
(eInspect 1,1028): tj 0x8000350
#0  0x70001260:0 in main (argc=0, argv=0x60000)
    at \SIERRA.$OS.VIVTEST.STEST12:32
#1  0x70001a20:0 in _MAIN () at \SPEEDY.$DATA06.T8432H01.CPLMAINC:68
```

- To obtain the address of the `ucontext` buffer and then trace the stack relative to the `ucontext` buffer:

```
(eInspect 1,1028): info locals
ucp = (ucontext_t *) 0x6fffede0
(eInspect 1,1028): tu 0x6fffede0
#0  0x70000cd0:0 in func3 (p=1879043416, q=0, r=134222120)
    at \SIERRA.$OS.VIVTEST.STEST12:21.030
#1  0x70000e50:0 in func2 (p=5, q=10) at \SIERRA.$OS.VIVTEST.STEST12:21.060

#2  0x70000fa0:0 in func1 (p=5) at \SIERRA.$OS.VIVTEST.STEST12:21.1
#3  0x700012d0:0 in main (argc=1, argv=0x8003010)
    at \SIERRA.$OS.VIVTEST.STEST12:33.1
#4  0x70001a20:0 in _MAIN () at \SPEEDY.$DATA06.T8432H01.CPLMAINC:68
```

tn (bt) Command

Prints a backtrace of the stack frames.

The `bt` command is an alias for the `tn` command. See bt (tn) Command (page 69) .

```
tn
```

unload-symbol-file Command

Unloads all symbol data associated with a specified loadfile name.

- Entered with no arguments, prompts you before unloading all symbol data having a per-process scope associated with the current process.
- Entered with only the `-g` option, prompts you before unloading all symbol files having global scope.

`unload-symbol-file [-g] `*`symbol-file-name`*

Where:

`-g`
: Unloads a symbol file with global scope.

`symbol-file-name`
: The name of a loadfile whose symbol information you want unloaded.

Related Command: symbol-file (symbol) Command, add-symbol-file Command

unmap-source-name (unmap) Command

Deletes previously set mappings for source file names. The command `unmap` is an alias for `unmap-source-name`.

Related commands: Use the `map` command to list the existing source file name mappings.

`{unmap-source-name|unmap} `*`map-number`*

Where:

`map-number`
: The number associated with a specific source file name mapping as displayed in the output of a map-source-name (map) Command.

This command deletes the mapping entry with number `map-number`. If `map-number` is the number of a source file name prefix mapping, than all mapping entries created as a result of this prefix are also deleted.

Examples

This example shows that when you delete a mapping for a pathname prefix that has resulted in the creation of new aliases, all such aliases are deleted.

- To display existing mappings:

```
(eInspect 3,-2): map

4. D:\usr\T1000\src\cpu\mips\x.c is aliased to /v/src/src/cpu/mips/x/c (Prefix match)
3. D:\usr\T1000 is aliased to /v/src
2. D:\usr\T1000\src\cpu\mips\x.c is aliased to /h/src/src/cpu/mips/x/c (Prefix match)
1. D:\usr\T1000 is aliased to /h/src
```

- To delete alias 1:

```
(eInspect 3, -2): unmap 1
```

- To display remaining aliases:

```
(eInspect 3, -2): map

4. D:\usr\T1000\src\cpu\mips\x.c is aliased to /v/src/src/cpu/mips/x/c (Prefix match)
3. D:\usr\T1000 is aliased to /v/src
```

  In the preceding example, alias 2 was also deleted because it depended on alias 1.

until Command

Continues executing the current process until a specified location is reached, a debugging event occurs, or the current stack frame returns.

`until [`*`locspec`*`]`

Where:

`locspec`

The location where you want execution to stop. See Syntax of locspec (page 59).

If you enter the `until` command with no arguments, the results are similar to those of the next command, except that at the bottom of a loop the until command steps through all remaining iterations.

Examples

- To execute until the function returns:

```
(eInspect 1,329): until
    -  building PCBReadyList: 0, 2, 4
  379                  PCBList_add( &PCBReadyList, PCBList.entry[0] );
```

- To execute until a specified location (line 382):

```
(eInspect 1,329): until 382
pcbDataStructs_initialize () at \SIERRA.$YOSE1.SYMBAT1.SCXXTST:382
  382                  pcb->flags.item.isReady = 1;
```

up (up-silently) Command

Selects the stack frame that calls the currently selected stack frame. The selected stack frame becomes the stack frame relative to which program state is displayed. The up command also prints information about the newly selected stack frame. Related commands are the down (down-silently) Command.

`{up|up-silently}` *`count`*

Where:

`count`

The number of frames to advance before selecting a stack frame.

Example

- Use the up and down commands and C sources as follows:

```
(eInspect 4,770): bt
 #0  pcbDataStructs_initialize () at \SIERRA.$YOSE1.SYMBAT1.SCXXTST:362
 #1  0x700016a0:0 in main (argc=1, argv=0x8003010)
     at \SIERRA.$YOSE1.SYMBAT1.SCXXTST:217
 #2  0x700011f0:0 in _MAIN () at \SPEEDY.$RLSE.T8432H01.CPLMAINC:68
 (eInspect 4,770): up
 #1  0x700016a0:0 in main (argc=1, argv=0x8003010)
     at \SIERRA.$YOSE1.SYMBAT1.SCXXTST:217
   217  pcbDataStructs_initialize();
 (eInspect 4,770): up
 #2  0x700011f0:0 in _MAIN () at \SPEEDY.$RLSE.T8432H01.CPLMAINC:68
 \SPEEDY.$RLSE.T8432H01.CPLMAINC:68: No such file or directory
 (eInspect 4,770): down
 #1  0x700016a0:0 in main (argc=1, argv=0x8003010)
     at \SIERRA.$YOSE1.SYMBAT1.SCXXTST:217
   217  pcbDataStructs_initialize();
 (eInspect 4,770): down
 #0  pcbDataStructs_initialize () at \SIERRA.$YOSE1.SYMBAT1.SCXXTST:362
   362  PCB_addAttribute( pcb, PCBAttribute_createSystem( 5 ) );
```

- Use the up and down commands for COBOL sources as follows:

```
(eInspect 7,416): bt
#0 EMMA.LISA () at \SIERRA.$AUDIT.DIVA.CBINIT:155
#1 0X70006950:0 in EMMA () at \SIERRA.$AUDIT.DIVA.CBINIT:136
#2 0x70003110:0 in EMMA () at \SIERRA.$AUDIT.DIVA.CBINIT:53
```

```
(eInspect 7,416): up
#1 0x70006950:0 in EMMA () at \SIERRA.$AUDIT.DIVA.CBINIT:136
 136          Call "Lisa".
(eInspect 7,416): up
#2 0x70003110:0 in MAIN () at \SIERRA.$AUDIT.DIVA.CBINIT:53
 53           Call "Emma".
(eInspect 7,416): down
#1 0x70006950:0 in EMMA () at \SIERRA.$AUDIT.DIVA.CBINIT:136
 136          Call "Lisa".
(eInspect 7,416): down
#0 EMMA.LISA () at \SIERRA.$AUDIT.DIVA.CBINIT:155
 155          DISPLAY "In Lisa".
```

vector Command

Changes the process that is designated as the current process when you are debugging more than one process. The current process is the process to which debugger commands apply.

`vector [`*`pin`*`] | [$`*`process-name`*`]`

Where:

`pin`

The process ID or process number of the process that you want to designate as the current debugging process. The command fails if you specify an invalid or nonexistent *`pin`*. Do not include the CPU number. For example, the *`pin`* for process `3,248` is `248`.

`$`*`process-name`*

The name of the process or process-pair you want to designate as the current debugging process. The command fails if you specify an invalid or nonexistent process name.

Considerations

Consider the following when using the `vector` command:

- Use the info Command with the `sessions` option to list the sessions running under one instance of Native Inspect. Use this command during multiprocess debugging.
- The `vector` command preserves debugging attributes you have set for the previously designated process.
- When privileged debugging is enabled in Native Inspect, use the `vector` command to view the state of any process in the CPU without having to attach to the process.

version Option

A command-line option that displays the versions of Native Inspect, GDB, and the Tool Command Language (Tcl). The show command with the version option displays the same output. (See show Command (page 116)).

`-- version`

vq Command

Displays information about process segments or, if you specify a *`segid`*, changes the selectable segment viewed by Native Inspect.

`vq [`*`segid`*`]`

Where:

`segid`

The segment ID for the segment about which to display information.

The segment corresponding to *segid* becomes the current in-use segment (marked with the letter g) for Native Inspect commands. (The current in-use segment for the process itself is marked with the letter p.)

If you do not specify a *segid*, the vq command displays information about the extended segments allocated by the current process.

wait Command

Instructs Native Inspect to stop prompting and wait for a Debug event to occur or for you to press the **Break** key.

You might need to use the wait command when debugging multiple processes because debugging events for processes other than the current process are reported only when Native Inspect waits (implicitly or explicitly) for events for the current process.

```
wait
```

whatis Command

Prints the data type of a specified expression.

whatis *expression*

Where:

expression

An expression used in the current debugging target process.

The whatis command is similar to the ptype Command, except that ptype prints detailed information about the data type, and whatis prints just the name of the data type.

which Command

The which command prints file, function, and line information of the specified (text or data) symbol.

which *symbol*

Where:

symbol

Specifies the symbol for which the data is retrieved. See symbol-file (symbol) Command (page 119).

Example

```
(eInspect): which main
Line 1346 of "myprog.c"
   starts at address 0x4000000000161720 <main>
   and ends at 0x4000000000161740 <main+0x20>.
```

x Command

Examines memory in any of several formats, independently of your program’s data types.

x [*/format*] *address*

Where:

/format

Specifies three optional specifications for format, size, and count of memory elements you want to display. See Syntax of /format (page 63).

The count must be first, but you can specify the format and size specifications in any order as follows:

format

The following options are supported:

- `a` – address
- `c` – char
- `d` – signed decimal
- `f` – float
- `i` – instruction
- `o` – octal
- `s` – string
- `t` – binary
- `u` – unsigned decimal
- `x` – hexadecimal

size

The following options are supported:

- `b` – byte
- `h` – half word (16 bits)
- `w` – word (32 bits)
- `g` – giant (64 bits or 8 bytes)

address

The address in memory at which you want the display to start.

For additional details regarding COBOL programs, see Chapter 3: Using Native Inspect With COBOL Programs.

Default Values

The defaulting rules for arguments to the `x` command enable you to easily display successive memory ranges using the same formatting:

- Default address – If you do not specify an address, the `x` command displays memory following the last address examined, or 0 if no address was previously examined.
- Default format – If you do not specify format, the `x` command uses the format specifications (format, size, and number) that you most recently specified with the `x` command.

These defaults (the last address examined, and the format specification) are set by a number of commands—the `x` command, the `info breakpoints` command, the `info line` command, and the print Command when you use it to display memory.

Convenience Variables $_ and $__

The convenience variables $_ and $__ store information about the most recent `x` command. The convenience variable $_ is automatically set by the `x` command to the last address examined, and $__ stores the contents of that address formatted as specified in the command.

Repeating the Last x Command

After you enter an `x` command, you can repeat it by pressing the **Enter** key at the next Native Inspect prompt. The effect is the same as if you had entered another `x` command with no parameters: listing continues with the address following the last one listed. This ability to repeat continues until you enter any other Native Inspect command.

Examples

- To display memory at address `0x70000fc0` and then display three machine instructions starting at that address:

```
(eInspect 7,464): x 0x70000fc0
0x70000fc0:0 <func_1+416>:      0x0900cc68
(eInspect 7,464): x /3i 0x70000fc0
0x70000fc0:0 <func_1+416>:      [MMI]       st4 [r52]=r51
0x70000fc0:1 <func_1+422>:                  addl r53=192,r1
0x70000fc0:2 <func_1+428>:                  nop.i 0x0;;
```

- To display memory at an address:

```
(eInspect 0,794): x /20c 0x70000ca8
0x70000ca8 <__STRING$4>:        109 'm'         111 'o'         110 'n'         105
 'i'        116 't'         111 'o'         114 'r'         0 '\000'
0x70000cb0 <__STRING$6>:        102 'f'         114 'r'         101 'e'         100
 'd'        0 '\000'         0 '\000'         0 '\000'         0 '\000'
0x70000cb8 <__STRING$5>:        100 'd'         112 'p'         50 '2'         0 '\
000'
```

- To display memory referenced by a pointer:

```
(eInspect 0,794): x /5xw pcbAttribute
0x8004ee0:      0x00030000      0x00000005      0x00000000      0x00000000
0x8004ef0:      0x00000000
(eInspect 0,794): x /5b pcbAttribute
0x8004ee0:      0x00    0x03    0x00    0x00    0x00
```

- To display instructions relative to the current location:

```
(eInspect 3,663): x /5i $ip
0x70002bf0:0 <pcbDataStructs_initialize+2256>:  [MMI]    break.m 0x2a;;
0x70002bf0:1 <pcbDataStructs_initialize+2262>:           ld8 r35=[r34]
0x70002bf0:2 <pcbDataStructs_initialize+2268>:           nop.i 0x0;;
0x70002c00:0 <pcbDataStructs_initialize+2272>:  [MMI]    adds r36=4,r35;;
0x70002c00:1 <pcbDataStructs_initialize+2278>:           adds r37=44,r36
```

- Use the following commands to display data in t (binary), o (octal), d (signed decimal), u (unsigned decimal), and x (hexadecimal) format:

```
(eInspect 7,498): x 0x6ffffee0
0x6ffffee0:     0x00250000
(eInspect 7,498): x /t 0x6ffffee0
0x6ffffee0:     00000000001001010000000000000000
(eInspect 7,498): x /o 0x6ffffee0
0x6ffffee0:     011200000
(eInspect 7,498): x /d 0x6ffffee0
0x6ffffee0:     2424832
(eInspect 7,498): x /u 0x6ffffee0
0x6ffffee0:     2424832
(eInspect 7,498): x /x 0x6ffffee0
0x6ffffee0:     0x00250000
```

- Use the following command to specify the current default to continue displaying memory. When you enter a format specification and an address, the x command uses those values as the defaults until you change the specification:

```
(eInspect 6,1103): x /3i 0x70000ca8
0x70000ca8 <__STRING$2+24>:             data8 0x14c84a407a
0x70000cb2 <__STRING$2+34>:     [MII]       break.m 0x0
0x70000cb8 <__STRING$2+40>:             break.i 0x0
(eInspect 6,1103): x/2
0x70000cbe <__STRING$2+46>:             break.i 0x0
0x70000cc2 <__STRING$3+2>:      [MII]       data8 0x103932ba333
(eInspect 6,1103): x /7
0x70000cc8 <__STRING$3+8>:             (p16) cmp4.eq p7,p16=45,r25
0x70000cce <__STRING$3+14>:             data8 0xc8dcdec6ca
0x70000cd2 <__STRING$3+18>:     [MII]       data8 0x15b1b7b6311
```

```
0x70000cd8 <__STRING$3+24>:                  data8 0xf481a480b1
0x70000cde <__STRING$3+30>:                  data8 0x014c84a40
0x70000ce2 <__STRING$4+2>:      [-3-]        data8 0x1237b21037b
0x70000ce8 <__STRING$4+8>:                   data8 0x1cdc9959d
(eInspect 6,1103):  x
0x70000ce8 <__STRING$4+8>:                   data8 0x1cdc9959d
```

NOTE: Note that offsets are specified in hexadecimal.

- Use the convenience variables $_ and $__ as follows:

```
(eInspect 2,348): x $pc
0x703a4310:2 <main+402>:        -535231967
(eInspect 2,348): print $_
$4 = (examine_w_type *) 0x703a4310:2
(eInspect 2,348): print $__
$5 = -535231967
```

5 Using Tcl Scripting

Introduction to Tcl

Native Inspect includes Version 8.5 of the open-source Tool Command Language (Tcl), which enables you to develop macros that can automate debugging operations. You can write Tcl scripts made up of built-in Tcl commands, and you can write your own Tcl commands.

Tcl scripts enable you to extend the power and functionality of Native Inspect. Tcl can help you create custom debugging commands suited for a particular domain or application. For example, if you are developing a complex application, you could also provide Tcl commands that aid in debugging the application.

Tcl Version 8.5 is the base scripting engine for Native Inspect. Some new primitives have been added to facilitate management on the NonStop system.

Learning Tcl

For more information about Tcl, see the following resources:

- http://www.tcl.tk/
- http://www.tcl.tk/doc/compiler.html
- http://www.tcl.tk/scripting/primer.html
- http://www.tcl.tk/doc/styleGuide.pdf

Using Native Inspect Tcl Commands

The Tcl command-line interpreter is part of Native Inspect, but Native Inspect commands have priority over Tcl commands.

Pass-Through of Tcl Commands

Some commands, such as set and help, exist in both Native Inspect and Tcl. To use the Tcl version of such a command, you must "pass through" commands to Tcl by specifying Tcl on the Native Inspect command line, as follows:

```
(eInspect 3,301):  tcl tcl-command
```

Native Inspect Commands Implemented in Tcl

Native Inspect contains several commands that are implemented in Tcl. You do not need to use the Tcl `pass-through` command when using the following Native Inspect commands:

- `a` command
- `base` command
- `comment` command
- `d` command
- `disassemble` command
- `da` command
- `env` command
- `eq` command
- `fn` command
- `i` command
- `jump` command

- `modify` command
- `reg` command
- `tn` command

Loading a Tcl Script

To load a Tcl script, use the Tcl source command. For example, to run the script named `myTcl`, enter:

```
(eInspect 3,301):  tcl source myTcl
```

To run the script, enter the name of the Tcl script at the Native Inspect prompt.

Using Variables Defined in a Tcl Script

You can use variables defined in a Tcl script, such as `$amount`, after you run the Tcl script that contains the definition.

The Tcl interpreter treats all command arguments as Tcl scripts. The Tcl environment is persistent for each interactive session, so variables you create and values you set are retained. For example:

```
(eInspect 3,301): tcl set x 0xabcd
(eInspect 3,301): tcl puts $x
0xabcd
```

Programming Native Inspect Tcl Commands

The Tcl Style Guide (http://www.tcl.tk/doc/styleGuide.pdf) provides a structure for Tcl script headers and the layout of package namespaces. The structure consists of the following:

- File header
  - Abstract
  - Copyright notice
  - Revision string
  - Package definition (package name, namespace, version)
- Procedure headers (one or more)
  - Abstract
  - Arguments
  - Results

Namespaces and Package Loading Rules

Tcl supports packages and hierarchical namespaces.

Creating Packages

Packages are libraries of Tcl code that you can create using the Tcl package provide command:

```
# mySub.tcl
package provide mySub 1.0
# my package code
```

Putting a Package in a Namespace

The global namespace contains the built-in Tcl commands, such as set, puts, and open. You should create your Tcl packages in your own child namespace, not in the global namespace, which should be the exclusive property of the application.

For example, to put package code in a namespace, use the `namespace eval` command. This example puts a package in a namespace of the same name:

```
# mySub.tcl
package provide mySub 1.0
namespace eval ::mySub:: {
   # my package code
}
#  More code
```

In this example, the namespace `::mySub::` contains the package `mySub`.

When your package is installed on a system, Tcl scripts can use your package by referring to its namespace, not its actual location.

It is important to create your packages in your own namespaces to avoid conflicts. Each exported subsystem command must be referenced by namespace.

Other Scripts Must Explicitly Require Your Package:

Any script that uses your package must explicitly require the package, as follows:

```
# In an app or another packagepackage require mySub
```

Explicitly Export Your Public Commands: Your package should export its public commands (but not its private commands), as follows:

```
namespace eval ::mySub::: {
   namespace export {[a-z]*}
}
proc ::mySub::myPublicProc{} {...}
```

Each namespace exports symbols that can be explicitly included, and unambiguously defined. The definition can be nested in the namespace itself (without the namespace qualifier) or declared as a member of the namespace, with the namespace name qualified in the proc name, as follows:

```
#---------------------------------------------------------
# Syntax:  dcba <context> <cradr> [ <count> ]
#--------------------------------------------------------------
proc ::mySub::dcba { context cradr {count 1} } {
   ...
}
...
```

If a Tcl script is to use the commands in mySub, the caller need only reference the package as follows:

```
package require mySub
```

Other Scripts Must Import Packages and Commands

If you have explicitly exported your public commands, another user can use a wild card to import your public commands, as follows:

```
#  In an app or another package
namespace eval ::mySub:: {
   namespace import ::mySub::*
}
```

A user can invoke your help by entering the `Tcl help` (or `tclhelp`) command, as follows:

`tcl help` *`commandname`*

Tcl Examples

- The code for the Native Inspect `eq` command is implemented in Tcl:

  ```
  # Syntax: eq <expr>
  #
  # Effect: Evaluate the expression and display the result in
  #         different formats.
  #
  #------------------------------------------------------------
  ```

```
proc eq { args } {
    # sanity check
    if { ! [llength args] } {
    SYNTAX_ERROR eq
    }
    set result [matheval $args]
    set char [ASCII $result]

    PUT "\n"
    PUT "OCT: [format %06o $result]  DEC: [format %-5d $result]
         HEX: 0x[format %04x result]  ASCII: \'$char\'\n"

}
proc empty_str {str} {
    expr { [scan $str "%s" tmp] == -1 }
#--------------------------------------------------------------
}
proc crunch_number {s endnum} {
    return [string match {[-0-9#%]} [string index $s 0]]
}
```

- To create and use a simple Tcl command (`allbases`):

```
file: mytcl
#-----------------------------------------------------------
# Syntax: allbases <expr>
# Effect: Evaluate the expression and display the result in
#         different formats.
#-----------------------------------------------------------
proc allbases { args } {

    # sanity check
    if { ! [llength args] } {
        SYNTAX_ERROR eq
    }
    set result [matheval $args]
    set char [ASCII $result]
    PUT "\n"
    PUT "OCT: [format %06o $result]
         DEC: [format %-5d $result]  HEX: 0x[format
%04x $result]  ASCII: \'$char\'\n"
}

(eInspect 3,615):tcl source mytcl
(eInspect 3,615):allbases 304

OCT: 000460  DEC: 304    HEX: 0x0130  ASCII: '...0'
(eInspect 3,615):
```

- To use a Tcl command (`ListPCBs`) to walk data structure (`symexpr` is a Tcl command used to evaluate symbolic expressions):

```
#-------------------------------------------------------------------
 # Syntax: ListPCBS
 # Synopsis: Walk the PCBList, printing info about each PCB
 #-------------------------------------------------------------------
 proc ListPCBs { } {

     set pcbCount [FORMAT [SYMEXPR PCBList.count] short DEC]
     PUT "\n$pcbCount Active PCBs\n"
     PUT "pin\tflags\tattributes\n"

     for { set i 0 } { [expr $i < $pcbCount] } { incr i } {
       set pin   [SYMEXPR PCBList.entry\[$i\]->ref.pcb->pin]
       set flags [SYMEXPR PCBList.entry\[$i\]->ref.pcb->flags.word]
       set attributeCount [SYMEXPR PCBList.entry\[$i\]->ref.pcb->attributeCount]
       PUT "[FORMAT $pin short DEC]\t[FORMAT $flags short]\
            t[FORMAT $attributeCount short DEC]\n"
```

```
        }
    }
```

- To use the Tcl script `ListPCBs` to walk data structure:

```
(eInspect 0,519): tcl source mypcb
(eInspect 0,519): ListPCBs
20      Active PCBs
pin      flags   attributes
0        0x6000  7
1        0x0500  0
2        0x6000  0
3        0x0500  0
4        0x6000  0
5        0x0500  0
6        0x0000  0
7        0x0000  0
8        0x0000  0
9        0x0000  0
10       0x0000  0
11       0x0000  0
12       0x0000  0
13       0x0000  0
14       0x0000  0
15       0x0000  0
16       0x0000  0
17       0x0000  0
18       0x0000  0
19       0x0000  0
```

Tcl Commands Provided by Native Inspect

Native Inspect includes the built-in commands that are provided by open-source Tcl (such as package provide and namespace eval). In addition to these, the commands listed in Table 14 are provided specifically for the HP NonStop platform.

NOTE: Starting with Tcl 8.5, the `expr`*`expression`* and `format`*`expression`* commands are enhanced to support evaluating and formatting of 64-bit expressions.

NOTE: Starting with Tcl 8.5, the `lexpr`*`expression`* command is no longer supported. The `mpexpr`*`expression`* and `mpformat`*`expression`* commands are deprecated.

Table 14 Tcl Commands for HP NonStop Systems

Tcl Command	Description
`expr`*`expression`*	Evaluates a 32-bit expression. Also supports 64-bit expressions.
`format`*`expression`*	Outputs a formatted 32-bit expression. Also supports 64-bit expressions.
`lexpr`*`expression`*	Evaluates a long expression (both 64-bit arithmetic and logical expressions). For `expression`, specify the high and low 32 bits separated by a space.
`mpexpr`*`expression`*	Evaluates a multiple precision expression. Can have an arbitrary operand. There is no automatic truncation, so be careful when performing bit-shifting operations.
`mpformat`*`expression`*	Outputs a formatted multiple-precision value. Example: `tcl puts (mpformat 0x%x [mpexpr 0xffffffff00000000]]` `0x0xffffffff00000000`
`{PUT|puts}`*`expression`*	Displays output; `PUT` does not include a newline, while `puts` does. `PUT` output is also captured to a log file if you use the Native Inspect `log` command to record a debug session.
`symexpr`	Passes the results of a symbolic expression to the Tcl script. Typically used to assign a return value to a Tcl variable. Is essentially the output of the Native Inspect `print`

Table 14 Tcl Commands for HP NonStop Systems *(continued)*

Tcl Command	Description

	command passed to the Tcl result buffer. The result might be a single value or multiple values.
	
	C++ Example:
	`(eInspect 3,301): tcl set x [symexpr NUMCPLUS`
	`(eInspect 3,301): tcl puts $x 15`
	
	COBOL Example:
	`(eInspect 5,1174): put [symexpr DI NOT EQUAL 0 1`
	`(eInspect 5,1174): tcl set x [symexpr DI IS GREATER THAN 5]`
	`(eInspect 5,1174): tcl puts $x 1`

A Command Mapping With Debug and Inspect

- Table 15 lists Debug commands and equivalent Native Inspect commands.
- Table 16 lists Inspect commands and equivalent Native Inspect commands.

Table 15 Debug Commands and Equivalent Native Inspect Commands

Debug Command	Equivalent Native Inspect Command
A	a (ASCII)
AMAP	amap
BASE	base
BM	mab (memory access breakpoint)
C, CM	delete
D, DN	d (display memory)
EXIT	exit
	quit
FC	fc (fix command)
FN, FNL	fn (find number)
HELP	help
	help option
I, IN	i (instructions)
IH	ih (info handlers)
INSPECT	switch
LMAP	info with symbol option
	disassemble
	da
MH	mh (modify handlers)
MODIFY	modify
	set (variable)
R	continue
STOP	kill
T, TN	bt
TJ	tj (trace from a jump buffer or from ucontext)
V	attach
	vector
VQ	vq (selects a segment)
=	eq (equals)
?	env (environment)

Table 16 Inspect Commands and Equivalent Native Inspect Commands

Inspect Command	Equivalent Native Inspect Command
Low-Level Inspect Commands	
`A`	`a` (ASCII)
`B`	`break`, `tbreak`
`BM`	`mab` (memory access breakpoint)
`C`, `CM`	`delete`, `dmab`
`D`	`d` (display memory)
`FC`	`fc` (fix command)
`F`	NOT SUPPORTED
`FN`	`fn` (find number)
`HIGH`	NOT APPLICABLE
`I`	`i` (instructions)
`M`	`modify`, `set` (variable)
`P`	NOT SUPPORTED
`R`	`continue`
`STOP`	`kill`
`T`	`bt` (`bt`, `tn`)
`VQ`	`vq` (select segment)
`?`	`env` (environment)
`=`	`eq` (equals)
High-Level Inspect commands	
`add alias`	NOT SUPPORTED
`add key`	NOT SUPPORTED
`add program`*`process`*	`attach`, `vector`
`add program`*`snapshot`*	`snapshot`
`add source assign`	`dir` (directory for source), `map-source-name` (base file name)
`break`	`break`, `tbreak` (temporary)
`clear`	`delete`
`comment`	`comment`
`delete alias`	NOT SUPPORTED
`delete keys`	NOT SUPPORTED
`delete source assign`	NOT SUPPORTED
`delete source open`	NOT SUPPORTED
`display`	`print`, `output`
`env`	`env` (environment)
`exit`	`exit`, `quit`

Table 16 Inspect Commands and Equivalent Native Inspect Commands *(continued)*

Inspect Command	Equivalent Native Inspect Command
fa	NOT SUPPORTED
fb	NOT SUPPORTED
fc	fc (fix command)
files	NOT SUPPORTED
fk	NOT SUPPORTED
if	condition
help	help, help option
history	show with commands option
hold	hold
icode	disassemble, da
identifier	ptype
info identifier	ptype
info location	NOT SUPPORTED
info objectfiles	info with dll option
info opens	NOT SUPPORTED
info savefile	NOT SUPPORTED
info scope	info with frame option
info segments	vq
info signals	ih
key	NOT SUPPORTED
list alias	NOT SUPPORTED
list breakpoints	info with breakpoints option
list history	show
list key	NOT SUPPORTED
list program	info with sessions option
list source assign	NOT SUPPORTED
list source opens	NOT SUPPORTED
log	log
low	NOT APPLICABLE
match scope	info with func option
modify	p, modify
modify signal	mh
obey	source
object	info with dll option
opens	NOT SUPPORTED

Table 16 Inspect Commands and Equivalent Native Inspect Commands *(continued)*

Inspect Command	Equivalent Native Inspect Command
`out`	`log`
`pause`	`wait`
`program`	`vector`
`resume`	`continue`
`save`	`save`
`scope`	`frame`, `select-frame`
`select debugger debug`	`switch` with `INSPECT` option
`select segment`	`vq`
`select language`	`set` (environment) with `lang` option
`select program`	`vector`
`select source system`	NOT SUPPORTED
`select systype`	NOT SUPPORTED
`set`	`set` (environment), `set` (variable)
`show`	`show`
`signals`	`ih` (info handler)
`source`	`list`
`source assign`	`map-source-name`
`step`	`step`, `stepi`, `next`, `nexti`
`stop`	`kill`
`system`	NOT SUPPORTED
`term`	NOT SUPPORTED
`time`	NOT SUPPORTED
`trace`	`bt` (`bt`, `tn`)
`volume`	NOT SUPPORTED
`xc`	NOT SUPPORTED

B Redirected and Aliased WDB Debugger Commands

The following WDB commands have a corresponding Native Inspect command with the relationship between the commands taking one of the following forms:

- A command alias – The WDB and Native Inspect syntax and semantics are identical. You can specify the WDB command and it automatically maps to the corresponding Native Inspect command.
- A command redirect – The WDB and Native Inspect syntax and semantics differ but the function is similar. The WDB command is disabled and the session displays a message redirecting you to use the appropriate Native Inspect command.

Table 17 lists the redirected and aliased WDB commands.

Table 17 Redirected and Aliased WDB Commands

WDB Command	WDB Function	Redirect or Alias	Native Inspect Command
`dumpcore`	Creates an HP-UX core file from a live process.	Redirect	`save` See: save Command (page 109)
`pathmap`	Define path mappings to locate object and source files.	Redirect	`map-source-name` See: map-source-name (map) Command (page 99)
`interrupt`	Interrupts the execution of the debugged program.	Alias	`hold` See: hold Command (page 84)
`redirect`	Controls command and output logging.	Redirect	`log` See: log Command (page 97)

Index