

Migrating from Inspect to Native Inspect

Revised for H06.08
Seth Hawthorne
NonStop Enterprise Division
Hewlett-Packard Company

Introduction

Native Inspect is the standard command-line debugger for debugging natively compiled programs on HP Integrity NonStop systems. It fulfills the roles formerly played by DEBUG and Inspect on S-series and TNS systems.

If you are reading this, you are probably migrating to Native Inspect for the first time. If you are familiar with the GNU *GDB* debugger, a de facto industry standard debugger found on many platforms, you will be able to get started with *GDB*-based Native Inspect in no time. Just scan for the notes that identify *GDB* differences.

If you are a long-time Inspect user, use the information in this document to translate common Inspect debugging operations to Native Inspect. The underlying concepts are the same, though expressed differently. If you are accustomed to Inspect commands, the biggest challenge may be retraining your fingers to issue the new commands that you learn.

Note: Before H06.06, Native Inspect did not support TNS/E COBOL; it now supports the TNS/E versions of COBOL, C/C++, and pTAL.

In this document, Native Inspect commands appear in **courier bold** and command parameters appear in *times italics*. You can abbreviate most commands.

Paradigm Shift

Accurate expectations are a key success factor in many endeavors, especially in this case. When you migrate to Native Inspect, keep a few things in mind:

- Native Inspect is a NonStop implementation of the GNU *GDB* debugger, **not** an Inspect descendent.
- Native Inspect is in its initial releases and will evolve.
- Inspect is a scope-based debugger in that code and data locations are identified relative to the containing procedure or function; Native Inspect uses source file names and line numbers.

Preparing to Debug

Compiling Programs

You can compile programs on the NonStop operating system or in a PC-based cross-development environment. In the latter case, you must transfer source files to the NonStop system when debugging and might have to define the location of source files to the debugger.

As on S-series systems, the following compilation optimization levels are supported:

Level	Description
O0	Limited optimizations and best debugging.
O1 (default)	Reasonable optimization and debugging, but some variables and locations might not be available.
O2	Compiled for performance with limited debugging.

During the development and debugging phase, compile programs at optimize level 0 (O0).

Tip: Use the `enof t lp * d` command to determine the optimization level of the functions that compose your program.

Starting Programs

You start programs under control of the debugger the same way as on earlier systems.

- **Guardian:** use the `rund` command.
- **OSS:** use the `-debug` command-line option

GDB Difference: GDB users commonly launch their programs from within the debugger, which has the benefit of breakpoints persisting across debugging sessions. Native Inspect does not support this capability at this time.

Debugger Selection Rules

The rules for selecting which debugger a process is delivered to have changed slightly:

- Processes are delivered to Visual Inspect if you have established a client connection with a matching user ID and either:
 - The process' `INSPECT ON` attribute is set.
 - The process is a TNS process.
- Otherwise:
 - TNS/E processes are delivered to Native Inspect.
 - TNS processes are delivered to Inspect.

Note: If the Inspect subsystem (`$IMON` and `$DMxx`) is not running, TNS processes are delivered to Native Inspect, but available debugging operations are limited to: tracing the stack, stopping the process, creating a snapshot file, or switching to Inspect once the Inspect subsystem is started.

Debugging Running Processes

From a TACL prompt, use the `DEBUG` or `DEBUGNOW` commands to force a program under debugger control.

Note: The same NonStop security rules are used to determine when the process can be delivered to the debugger, and the previously described debugger selection rules are applied.

Specify the `TERM` option to start the debugger on your current terminal rather than on the home terminal for the process.

From within Native Inspect, use the `attach` command to obtain control of a running process. For example, `attach 235`.

Inspect Difference: Native Inspect must run in the same CPU as the processes it is debugging.

Multiprocess Debugging

In Inspect, you could debug multiple processes with a single debugging session; however, you could only view the state of one process at a time, and you often had to use the break key to switch between processes.

Native Inspect support for multiprocess debugging is more limited in that all processes must execute in the same CPU.

Visual Inspect is the best tool for debugging multiple processes, because it allows you to view and easily switch among all processes. Alternatively, you can use several terminal sessions to run multiple instances of Native Inspect.


Listing Source

One of the first things you will likely want to do is list the source text surrounding the current location in the currently selected stack frame.

In Native Inspect, use the `list` command to list source. The first invocation lists lines surrounding the current location. The current location is marked with an asterisk. Subsequent invocations list following lines. To list source at a specified location, specify a line number and optional file name:

list *line-number*
list *file:line-number*

Inspect Difference: Native Inspect does not support the ability to search for text in source files or to close open source files.

 **Gotcha:** In Native Inspect, the `source` command has the same meaning as the Inspect `OBEY` command.

If Native Inspect cannot locate your source files, see *Locating Source and Symbols* on page 6.

To obtain information about source files, use these Native Inspect commands:

info source List information about current source file.
info sources List information about all accessed source files.

Tracing the Stack

In Native Inspect, use the `bt` (**back trace**) command to list the frames on the call stack.

When debugging, you often gain control of a process in a low-level function and then need to determine the circumstances that lead to it being called. To select the specified frame as the frame relative to which the debugger displays program state, use the command:

frame *frame-number*

This command serves the same function as the Inspect `SCOPE frame-number` command. Native Inspect also has `up` and `down` commands for selecting the preceding or succeeding stack frame.

Tip: If you forget the location at which your program is suspended, issue the `frame 0` command, abbreviated `fr 0`, to display the current execution location.

By default Native Inspect lists function arguments in the stack trace. The following command disables this behavior:

set print args off

To obtain detailed information about a stack frame, use the `info frame` command.

Controlling Program Execution

Native Inspect supports the same concepts as Inspect for stepping program execution but uses different command names to refer to the operations:

Inspect	Native Inspect
STEP [OVER]	next or n
STEP IN	step or s
STEP OUT	finish

Process-control commands are also different:

Inspect	Native Inspect
RESUME	continue or c
STOP	kill or k

Breakpoints

Native Inspect supports the same breakpoint concepts as Inspect, though they are written differently.

To set a breakpoint, use one of these forms of the `break` command, abbreviated `b`:

break *line-number* Set breakpoint on a line number.
break *file:line* Set a breakpoint on a line number in the specified file.
break *function* Set a breakpoint at the entry to a function.
break *label* Set a breakpoint at a code label.
break *paragraph* Set a breakpoint on a COBOL paragraph.
break *paragraph OF section* Set a breakpoint on a COBOL paragraph in the specified section.

Inspect Difference: Unlike Inspect, code location specifications are not prefixed with a #.

If there are multiple instances of a specified function, Native Inspect displays a selection menu.

To set a temporary breakpoint, use the `tbreak` command.


To list currently set breakpoints, use the **info breakpoints** command.

Like Inspect, breakpoints are identified to related commands using ordinal numbers. You can apply these commands:

- condition** *breakpoint-ord condition*
Add or remove a breakpoint condition.
- ignore** *breakpoint-ord*
Ignore the breakpoint a specified number of times (Inspect EVERY clause).
- commands** *breakpoint-ord*
Add a list of commands to be executed when the breakpoint is hit.

To delete breakpoints, use one of these commands:

- delete** *breakpoint-ord* Delete the specified breakpoint.
- delete *** Delete all breakpoints.

 **Gotcha:** Inspect uses the **c** (**clear**) command to clear breakpoints. Native Inspect interprets the **c** command as the **continue** command, which resumes process execution.

To set and clear memory-access breakpoints, use the **mab** and **dmab** commands, respectively.

Limitation: The memory-access breakpoint is not currently listed in the breakpoint list nor can a condition be applied to it.

To enable and disable breakpoints, use these commands:


- enable** *breakpoint-ord*
- disable** *breakpoint-ord*

To set a breakpoint that is triggered when a process ABENDs or stops, use these commands:

- catch abend**
- catch stop**

Displaying Variables

In Native Inspect, you use the **print** command to display a variable or the results of an expression. Unlike Inspect, Native Inspect does not have any constraints on the C/C++ expressions that can be evaluated. You can even perform assignments!

 **Gotcha:** Native Inspect has a **display** command, which adds a variable or expression to the list that is automatically displayed whenever the program is suspended. While useful, this behavior is different from the correspondingly named Inspect command.

Formatting Values

In Inspect, you used the **IN** clause to control the display radix. In Native Inspect, you must specify a */fmt* option to the **print** command, where *fmt* has one of these values:

<i>fmt</i>	Radix
a	Address
c	Character
d	Decimal
f	Float
o	Octal
t	Binary
u	Unsigned decimal
x	Hexadecimal

For example, **print /x ptr**

The Inspect **DISPLAY AS** command allowed you to format the memory associated with a variable using the type of another variable. This command was commonly used to format the contents of a raw message buffer using the type of the structure corresponding to the message. In Native Inspect, you must use the following GDB syntax:

print {type}variable

In Native Inspect, unlike Inspect, you can use C/C++ casts to change the type of a C/C++ or pTAL variable. For example,
print (char *) StrAddr

Pointers

Like GDB, Native Inspect automatically dereferences C/C++ character pointers (`char *`).

Limitation: Whereas pTAL automatically dereferences pointers, Native Inspect currently treats them like C/C++ non-character pointers; you must explicitly dereference them.

Arrays

Native Inspect does not directly support the Inspect `FOR` clause. To constrain the number of array elements displayed, use this command:

```
set print elements count
```

In C/C++ and pTAL programs, you can also use the GDB “array operator” to specify the subscript at which array display is to stop. For example,

```
print array@4
```

Native Inspect compresses repeated array elements, displaying the element value followed by the count. This command controls the number of values that must be repeated before they are compressed:

```
set print repeats count
```

Expressions

The Native Inspect `print` command accepts expressions in addition to variables. Unlike Inspect, you do not need to enclose expressions in parenthesis.

Variable Address

To obtain the address of a variable, use the C/C++ `&` operator or the `info address` command. In COBOL programs, you can use the `ADDRESS OF` operator.

Variable Type

Instead of the Inspect `INFO IDENTIFIER` command, Native Inspect provides several commands for obtaining type information about a variable:

```
ptype variable
whatis variable
```

Auto Display

In Inspect, you could use the `SET STATUS ACTION` command to execute display commands that automatically displayed “interesting” variables each time a program is suspended. In Native Inspect, use the `display` command to add a variable or expression to the “auto display list,” each item of which is displayed when the program is suspended.

Other useful commands:

<code>display</code>	List items on the auto display list.
<code>delete display ordinal</code>	Delete the specified item from the list.
<code>enable ordinal</code>	Enable display of the specified item.
<code>disable ordinal</code>	Disable display of the specified item.

New Commands

Native Inspect provides several capabilities not found in Inspect:

<code>info locals</code>	Display all local variables.
<code>info args</code>	Display arguments to the current stack frame.
<code>info variables pattern</code>	List all variables matching <i>pattern</i> .

Scoping


Native Inspect does not support Inspect’s ability to access static variables contained in inactive scopes (procedures or functions for which there is no instance on the call stack). Specifically, with Inspect you could use the `scope` command to select the scope or you could qualify the variable name with the name of the containing scope.

Modifying Variables

To modify a variable, use either of these commands:

```
print variable = value
set variable variable = value
```

The *value* can be a constant, another variable, or the result of an expression.

 **GDB Gotcha:** The **variable** clause of the **set** command is optional but must be specified if the name of the variable is the same as a debugger option, listed by the **show** command.

Locating Source and Symbols

Locating Source

If executable files or the source files compiled to create them are relocated, Native Inspect displays an error indicating that it cannot locate the source file at the compiled location recorded in the object file. You must then specify how to locate the file.

If the file was relocated to another subvolume or directory but has the same base name, use the command:

```
dir dir-or-subvol
```

Native Inspect searches that location for source files. When you specify the **dir** command with no argument, Native Inspect asks you if you want to clear the search path.

If the base filename has changed, which is common when moving source files from a cross-development environment to Guardian, you must use the **map** command to specify an alternative location:

```
map [[ original ] = current]
```

You can specify fully qualified *original* and *current* file names or (as of H06.07) you can specify file name prefixes. When you specify prefixes, Native Inspect attempts to match the *original* prefix against subsequently referenced source filenames. If there is a match, the *original* prefix is replaced by the *current* prefix and the file mapping is added to the file mapping list.

Omitting all arguments lists the contents of the file mapping list. You can omit the *current* original name when defining a mapping for the file from which you are currently listing source. Use the **umap** command to delete a mapping.

Inspect Difference: Native Inspect does not support automatic rules for mapping file-name extensions to Guardian file names.

Locating Symbols

When the development cycle is complete, loadfiles are commonly stripped of symbols to save disk space. When debugging such a program, you might notice this message:
(no debugging symbols found)...

You will also find that source file names and line numbers are missing in the stack trace and that you cannot reference variables or list source.

Whereas Inspect required you to specify alternative symbol files when adding or selecting a program, Native Inspect allows you to do so at any time during the debugging session. Use this command:

```
symbol-file filename
```

As with Inspect, you must make sure that the symbols correspond to the loadfile being debugged. (Native Inspect issues a warning if the compilation timestamps differ.)

Note: You will also often need to load symbol files when examining process snapshot files.

To list loaded symbol files, use this command:

```
info symbol-files
```

To delete a loaded symbol file, use this command:

```
unload-symbol-file
```

Note: For DLL loadfiles, Native Inspect uses the actual load address to address variables. If you need to override use of the actual address, the **add-symbol** command allows you to specify a base address when loading the file.

DLLS

Dynamic link libraries (DLLs) are standard on TNS/E systems. As a result, your program run-time image will be composed of system DLLs, language run-time DLLs, and any DLLs that you define.

To list information about the DLLs that compose your program, use this command:

```
info dll
```

To load missing symbols for any DLLs, use the **symbol-file** or **add-symbol-file** commands (see *Locating Symbols* on page 6).

As of H06.07, you can use the following commands to suspend program execution when a DLL is loaded or unloaded:

```
catch load [dllname]  
catch unload [dllname]
```

Data Segments

The **vq** command is similar in function to the Inspect **INFO SEGMENTS** command. When specified with no arguments, **vq** lists all data segments. Specifying a segment id as an argument changes the selectable segment relative to which Native Inspect displays process state (Inspect **SELECT SEGMENT**).

Snapshot Files

Snapshot files, also referred to as “save” or “save abend” files, store the user process state of a process for later “post-mortem” debugging.

The system automatically creates snapshot files when a process with the **SAVEABEND** attribute set calls **ABEND** to terminate. To create a snapshot file in Native Inspect, use the **save** command.

To use Native Inspect to examine a TNS/E snapshot file, start Native Inspect and use this command to load the file:

```
snapshot filename
```

Note: You will likely need to load symbols and might need to point to alternative source file locations. For more details, see *Locating Source and Symbols* on page 6.

Note: You can only examine one snapshot file at a time with an eInspect session. To examine multiple files at the same time, run eInspect in multiple terminal sessions.

Scripting

Inspect scripting capabilities were limited to placing sequences of commands in a file that you could then execute using the **OBEY** command.

Like Inspect, Native Inspect allows you to execute commands from a file. Unfortunately, the command to do so is named **source**, which has a much different meaning in Inspect.

The integrated TCL (Tool Control Language) interpreter in Native Inspect supports the development of sophisticated debugging scripts. For example:

File: myTCL:

```
proc allbases { args } {  
    set result [matheval $args]  
    set char [ASCII $result]  
    PUT "\nOCT: [format %06o $result]"  
    PUT "DEC: [format %-5d $result]"  
    PUT "HEX: 0x[format %04x $result]"  
    PUT "ASCII: \'$char\'\n"  
}
```

From Native Inspect:

```
(eInspect) tcl source mytcl  
(eInspect) allbases 304  
OCT: 000460 DEC: 304 HEX:  
0x0130 ASCII: '...0'
```

Machine-Level Debugging

Sometimes viewing your program execution at the source level might be insufficient, and instead you need to see how the machine is executing the compiler-translated instructions. Doing so reveals the much larger register file, longer instruction sequences, and compiler optimizations inherent to TNS/E processors.

Whereas Inspect supported a low-level mode for this purpose, machine-level debugging commands are integrated into Native Inspect.

Examining Memory

Use the “eXamine”, **x**, command to display memory at a specified address. It accepts an optional formatting clause and address:

```
x [[/NFU] address]
```

N is an optional count of the number of memory units to display.

F is an optional letter that controls the formatting of the memory. The following values are recognized in addition to the *fmt* letters recognized by the print command:


fmt	Radix
i	Instruction
s	String

U is an optional letter that controls the memory unit size to display:

Unit	Size
b	Byte
h	Half-word
w	Word
g	Giant (8 bytes)

For example,

```
x /10 0x8001ac0
x /5cb 0x8001ac0
```

 **GDB Gotcha:** When you do not specify a *format* or *size*, the last value specified to the **x** command is used. When you omit *address*, the address following the last displayed address is used.

To specify an address to the print command, use this syntax:

```
print *address
```

Listing Instructions

To list instructions corresponding to source lines, use the **disassemble** command. By default, it lists all instructions for the current function. You can specify an address range, however.

Tip: To determine the address range of a line, use the **info line** command.

To display instructions surrounding the current instruction pointer, it is often easier to use the **x** command. For example:

```
x /4i $ip
```

This command displays four instructions starting at the address contained in the Instruction Pointer register.

Inspect Difference: When a breakpoint is set, you will see the breakpoint rather than the instruction that it replaces.

Displaying Registers

A TNS/E processor has 128 general-purpose registers, 64 predicate registers, floating-point registers, and status registers. To see the key registers associated with the current stack frame, use this command:

```
info frame
```

To display all general-purpose and status registers, use this command:

```
info registers
```

To display all registers including floating-point registers, use this command:

```
info all-registers
```

To use a register value in an expression, prefix its name with a **\$**. For example:

```
print $gr4
```


Execution Control

The **step** and **next** commands have corresponding **stepi** and **nexti** commands for stepping instructions.

To set a breakpoint at an address, use this command:

```
b *address
```

Privileged Debugging

Like DEBUG or Inspect, you can use Native Inspect to perform privileged debugging. To enable privileged debugging, you must log on as super ID and issue this Native Inspect command:

```
priv on
```

Once enabled, you can examine privileged memory regions. You can also load and use symbols for privileged DLLs.

When privileged debugging mode is enabled, the **attach** command issues a DEBUGNOW request.

Switching Debuggers

Occasionally, you might need temporary functionality provided by another debugger or a different debugger than you started.

You can switch TNS/E native processes between Native Inspect and Visual Inspect, which provides a graphical user interface for debugging.

To switch a process from Native Inspect to Visual Inspect, establish a Visual Inspect connection and issue the Native Inspect **switch** command.


To switch a process from Visual Inspect to Native Inspect, select **Program > Switch to System Debugger**.

Convenience Commands

Inspect Difference: Native Inspect does not support Inspect convenience commands, such as: SET STATUS, SET PROMPT, ADD KEY, and so on.

Changing the Working Directory

Native Inspect uses the **cd** or **volume** command to change the current working directory used for resolving unqualified filenames. Use the **pwd** command to list the current working directory.

 **NonStop Gotcha:** When the debugger is started automatically, as a result of a **rund** command or process debug request, its working directory is your logon subvolume/directory NOT your current subvolume/directory.

Logging Session Output

To log the output of your debugging session, use this command:

```
log file
```

If the log file already exists, output is appended to it.

To disable logging, enter the command with no arguments.

Command History

Native Inspect maintains a history of the commands that you execute. To list command history, use this command:

```
show commands
```

To edit the previous or a specified command, use the **fc** (fix) command. To invoke a specified command, use **!command-number**.

As of H06.07, you can reissue the following commands by pressing the RETURN key: **list**, **finish**, **next**, **nexti**, **print**, **step**, **stepi**, and **x**.

Value History

Native Inspect saves displayed values in a value history list, which is analogous to the command history list. Entries are identified by a **\$**-prefixed ordinal number that is printed at the beginning of **print** command output. Values can be recalled and used with other commands. For example,

```
print $3
```

Using Convenience Variables

Value history variables are one example of GDB convenience variables, whose names are prefixed with a **\$**. To define variables to hold values commonly used in your debugging session, use the **set** command. For example:

```
set $curObj=obj->next->next
```

You can then use the variable name with other debugging commands. For example:

```
print $curObj->flags
```

Creating a Custom File

To automatically execute commands at the startup of each Native Inspect session, place them in this file in your logon default subvolume:

```
EINSCSTM
```

Controlling Output

By default, Native Inspect paginates output, prompting you after each screen page of output. To disable pagination, use the command:

```
set pagination off
```

Destructive commands often issue a confirmation prompt before performing their action. To disable the prompt, use this command:

```
set confirm off
```

Limitations

- Native Inspect does not yet support the ability to list information about a program's open files. (Inspect INFO OPENS).

- Native Inspect support for pTAL is currently limited. In many respects, pTAL is treated like C. You must use C-style pointer dereferencing and transform pTAL expressions to corresponding C expressions.
- Online help reflects GDB online help, which is not as complete as Inspect online help.
- Native Inspect does not support the following Inspect formatting capabilities:
 - PIC or FORMAT
 - SPI formatting
 - System types
- Native Inspect does not support convenience commands, such as defining function keys, displaying status line information, and so on.

Resources

The *Native Inspect Manual*, part number 528122-005, is available in the NonStop Technical Library at <http://www.hp.com/go/ntl>.

A wide variety of resources are available on the internet about GDB and WDB, the HP-UX variant. Native Inspect is currently based on GDB version 4.17.

- GDB Home
<http://www.gnu.org/software/gdb/>
- WDB Home
<http://www.hp.com/go/wdb>
- TCL Home
<http://www.tcl.tk/>
- GDB Quick Reference card
<http://refcards.com/refcards/gdb/gdb-refcard-letter.pdf>

Several books are also available on GDB, including an O'Reilly *GDB Pocket Reference*.

Appendix A - Inspect to Native Inspect Command Mapping

In the following table, | is used to identify alternative choices.

Inspect Command	Native Inspect Command	Description
A	a	Display memory in ASCII. (Native Inspect Debug compatibility command)
ADD ALIAS	Not supported	Add a command alias.
ADD KEY	Not supported	Add a command binding to a terminal function key.
ADD PROGRAM	attach snapshot	Debug a running process or load a snapshot file.
ADD SOURCE ASSIGN	dir map	Define how to locate a source file that is not at its compile-time location. (Note: Native Inspect does not support partial source assigns.)
B[REAK]	b[reak] tbreak	Set a breakpoint (or temporary breakpoint) at the specified location.
BD	mab	Set a memory access breakpoint.
CD	cd volume	Change the current directory, used to resolve file names that are not fully qualified.
C[LEAR]	delete	Delete a code breakpoint or memory access breakpoint.
DELETE ALIAS	Not supported	Delete command alias.
DELETE KEYS	Not supported	Delete function key binding.
DELETE SOURCE ASSIGN	Not supported	Delete a source file name mapping rule.
DELETE SOURCE OPEN	Not supported	Close an open source file so that another program can access it.
D[ISPLAY]	print	Print the value of a variable or result of an expression.
ENV	env	Display information about the debugging session.
EXIT	exit	Terminate the debugging session
FC	fc	Fix command; edit and reexecute a previous command.

Inspect Command	Native Inspect Command	Description
FILES	Not supported	Display information about files opened by the current process.
FA	Not supported	Edit a previously defined command alias.
FB	Not supported	Edit a previously defined breakpoint.
FX	Not supported	Edit a previously defined function key binding.
FN	fn	Find number; search memory for a value.
HELP	help	Display online help.
HIGH	N/A	Enter symbolic debugging mode
HISTORY	show commands	List the history of most recently executed commands.
HOLD	hold	Suspend execution of the currently running process.
ICODE	i da disassemble x	Display disassembled machine instructions.
IDENTIFIER	ptype whatis	Display type information about an identifier.
IF	condition if	Define a condition that must be true before a specified breakpoint is triggered.
INFO IDENTIFIER	ptype whatis	Display type information about an identifier.
INFO LOCATION	Not supported	Display addresses of compiler-defined statements.
INFO OBJECTFILES	info dll	Display information about the loadfiles of the current process.
INFO OPENS	Not supported	Display information about files opened by the current process.
INFO SAVEFILE	Not supported	Display information about a snapshot file.
INFO SCOPE	info frame	Display information about a stack frame or procedure/function/program unit.
INFO SEGMENTS	vq	Display information about the current process' data segments.

Inspect Command	Native Inspect Command	Description
INFO SIGNALS	ih	Display information about defined signal handlers.
KEY	Not supported	List defined function key bindings.
LIST ALIAS	Not supported	List command aliases.
LIST BREAKPOINTS	info breakpoints	List defined breakpoints. (Native Inspect Debug currently does not list the memory access breakpoint.)
LIST HISTORY	show commands	List the history of most recently executed commands.
LIST KEY	Not supported	List defined function key bindings.
LIST PROGRAM	info sessions	List all processes being debugged.
LIST SOURCE ASSIGN	map	List defined source file mapping rules.
LIST SOURCE OPENS	info source info sources	List information about opened source files.
LOG	log	Control logging of session input and output to a file.
LOW	N/A	Switch to machine-level debugging mode.
MATCH IDENTIFIER	info variable	List identifiers whose names match a specified pattern.
MATCH SCOPE	info func	List information about procedures/functions/program units whose names match a specified pattern.
M[ODIFY]	m set variable <i>var = value</i>	Modify the value of a variable.
MODIFY SIGNAL	mh	Modify a signal handler.
OBEY	source	Execute commands from the specified input file.
OBJECT	info dll	Display information about the current executable.
OPENS	Not supported	Display information about files opened by the current process.

Inspect Command	Native Inspect Command	Description
OUT	log	Log session output to a file. (Native Inspect does not support only logging output.)
PAUSE	wait	Suspend debugger prompting until a process debug event occurs, or the user presses the break key.
PROGRAM	vector	Select the specified program as the current program, on which all debugger operations apply.
RESUME	continue	Continue execution of the process.
SAVE	save	Save process state to a snapshot file for later “post-mortem” debugging.
SCOPE	fr[ame] select-frame	Select a specified stack frame as the frame relative to which program state is displayed.
SELECT DEBUGGER DEBUG	switch	Switch the current process to another debugger.
SELECT SEGMENT	vq	Select an extended data segment as the “current” segment relative to which the debugger displays program state.
SELECT LANGUAGE	set language	Set the current language used to parse and evaluate expressions. (Useful when debugging mixed-language programs.)
SELECT PROGRAM	vector	Select which program is the current program.
SELECT SOURCE SYSTEM	Not supported	Read source files from specified system.
SELECT SYSTYPE	Determined by the path type supplied to the last <code>cd</code> command.	Set whether file names are resolved using Guardian or OSS rules.
SET	set	Set a debugger option.
SET RADIX	base	Set default input or output radix. (Native Inspect Debug compatibility command)
SHOW	show	List debugger options and their values.
SIGNALS	ih	Display information about defined signal handlers.

Inspect Command	Native Inspect Command	Description
SOURCE	list	List program source text.
STEP IN	step, stepi,	Step program execution a statement or an instruction at a time, stepping in to function/procedure/program unit calls.
STEP OUT	finish	Step program execution out of the current function/procedure/program unit call.
STEP OVER	next, nexti	Step program execution a statement or an instruction at a time, stepping in to function/procedure/program unit calls.
STOP	kill	Terminate program execution.
SYSTEM	Not supported	Set the system to be used when resolving Guardian file names.
TERM	Not supported	Redirect terminal output to the specified terminal.
TIME	Not supported	Report the current time
TRACE	bt	List the frames on the program call stack.
VOLUME	Volume cd	Change the default Guardian volume used to resolve unqualified file names.
XC	!	Execute the specified command from the history list.